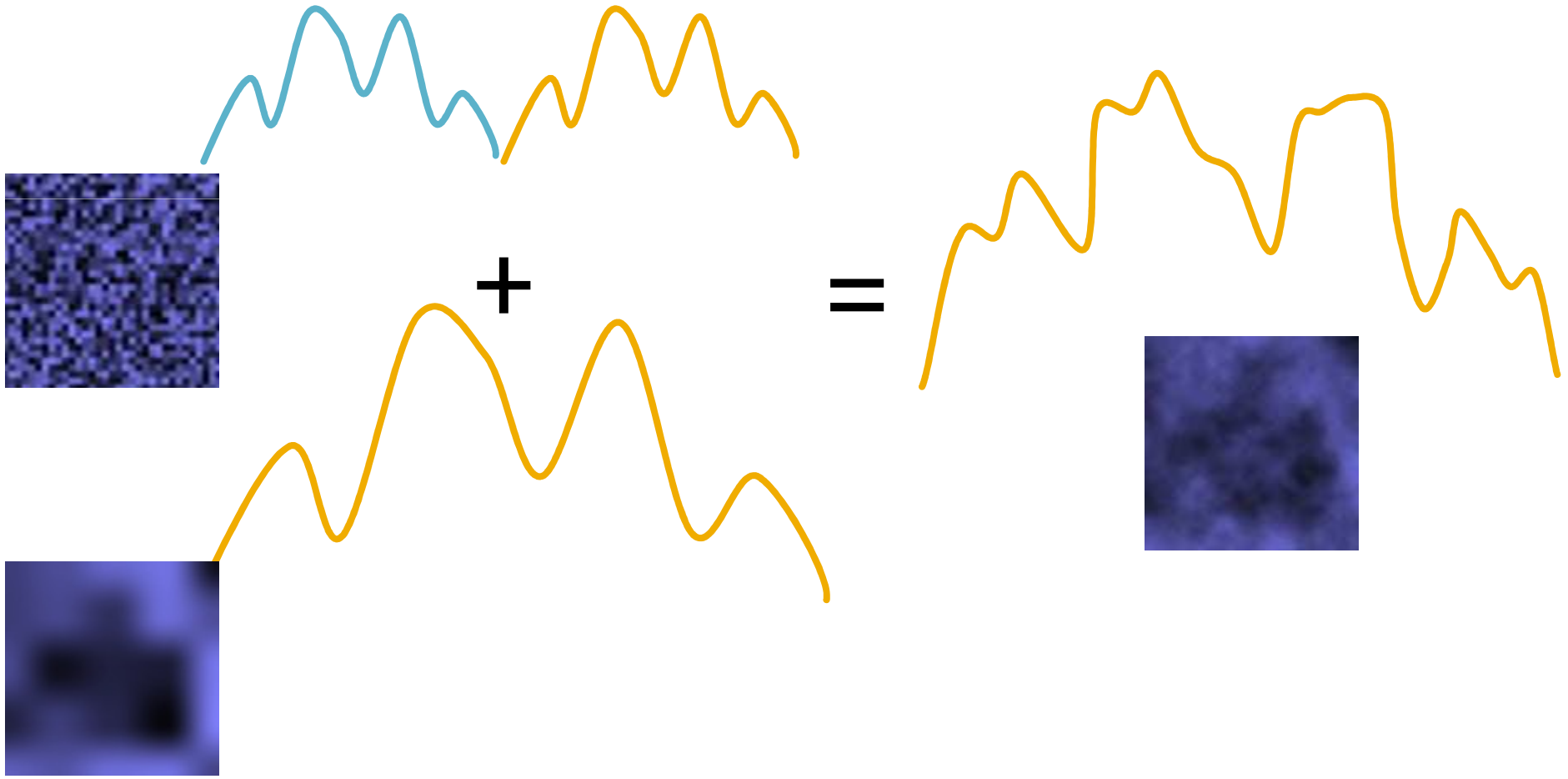
Rekurzív algoritmusok

Labirintus fő lépések

- Véletlen szám generálás
- Labirintus felépítése 1x1-es felbontástól a teljes méretig
- Labirintusban egy kiindulási pontból az összes pontba legrövidebb út keresése
- Egy végállomásból elindulva visszafejteni a legrövidebb utat a kiindulási állapotig

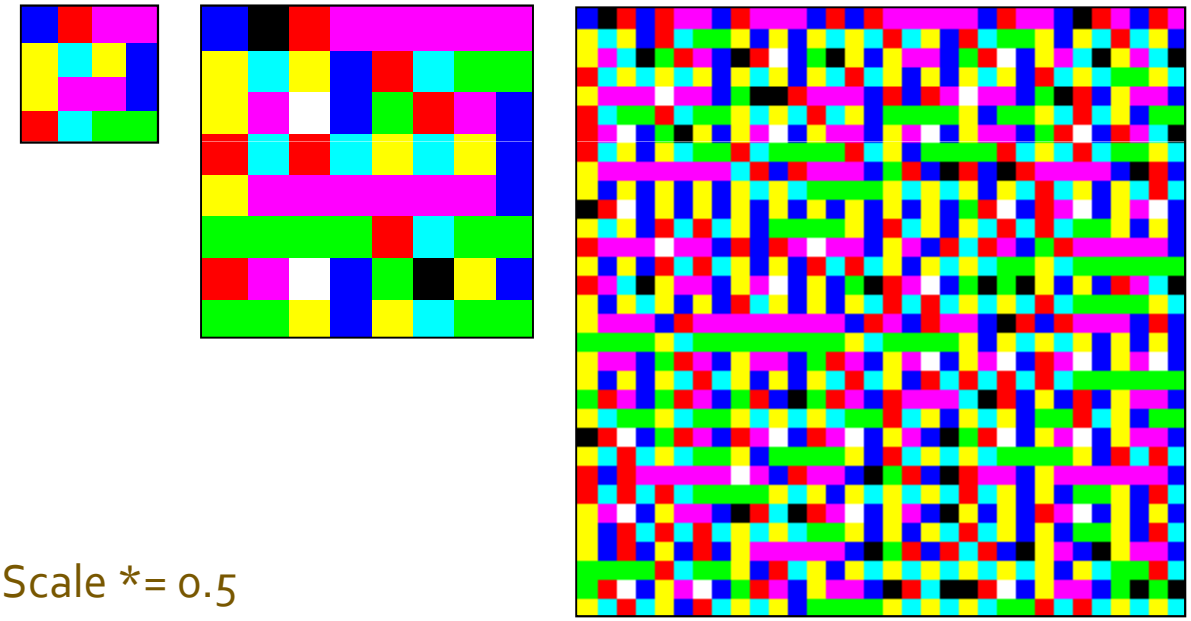
Véletlenszám

- Fraktál zaj



Véletlen szám generálás

- 2D zaj, 4 színcsatornában 4 független véletlen szám tárolható
- MipMap
  - glTexImage2D(GL_TEXTURE_2D, level, GL_RGBA32F, …);
  - glFramebufferTexture2D(GL_FRAMEBUFFER, …, TexID, mipLevel);



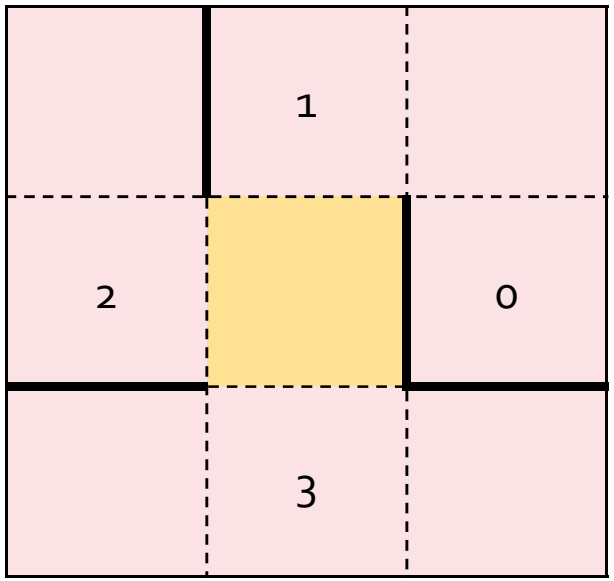
Brown mozgás:

- R kezdeti értéke 0.5
- noiseScale kezdeti értéke 0.5
- minden mipmap szintnél noiseScale *= 0.5

```
vec4 R = textureLod(inputTex2, fTexCoord, level + 1);  //előző szint
vec4 rand = texelFetch(noiseTex,  ivec2(mod(coord + ivec2(noiseSeed, 0),
                                           ivec2(16))), 0);
R = R + noiseScale * (rand - 0.5);
```

Labirintus generálás

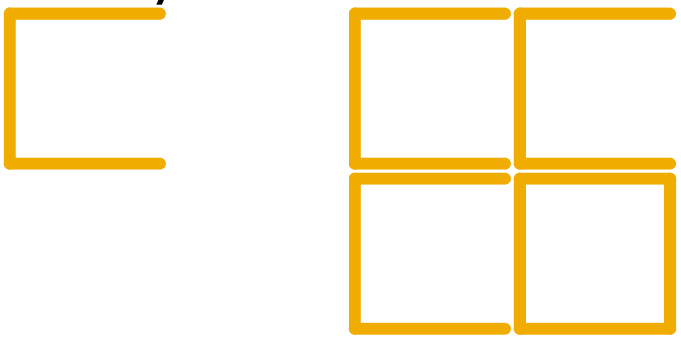
- Cella:
  - 4 szomszéd, 4 csatorna
  - Ha vezet út a szomszéd felé akkor 1, egyébként 0



V(R,G,B,A) = (0,1,1,1)

Fő lépések

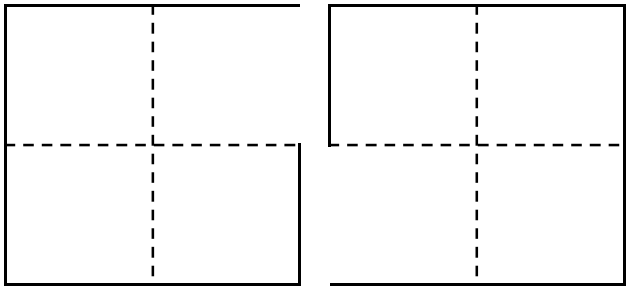
- Rekurzív tovább osztás
  - Szülő cella értékeit megörököljük
  - Közbülső falakat töröljük
  - Az átjárók egyik felét lezárjuk
  - Az új 4 cellát egy irányban felosztjuk (4 lehetséges irányból 1-et véletlenszerűen választunk)



Átjárók problémája

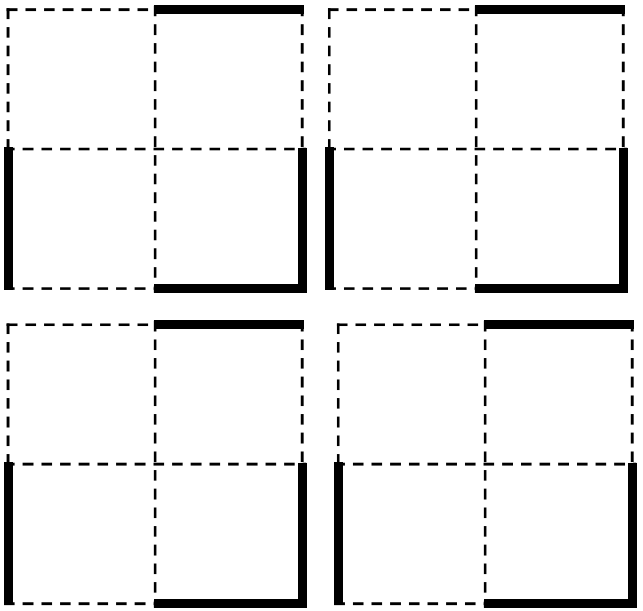
Probléma:

Két szomszédos cellacsoport
másképp dönt az átjáró lezárásáról
(kommunikáció kellene a szálak között)



Legegyszerűbb megoldás:

Mindig ugyanazt választjuk,
pl. bal oldalt és felül tartjuk
meg az átjárókat.
Ismételhető lesz.



Logikai operátorok

Cellák indexelése (index):

2	3
0	1

Irányok indexelése (csatornák):

	1	
2		0
	3	

Maszk (M), mely irányokban van szomszédja „befele”:

(1,0,0,1)	(0,0,1,1)
(1,1,0,0)	(0,1,1,0)

Maszk (D), mindig bal oldalon és felül tartjuk meg az átjárókat (melyik irányban kell átjárót lezárni):

(0,0,0,0)	(0,1,0,0)
(0,0,1,0)	(1,0,0,1)

Halmaz operátorok:
A * B: metszet
uni(A, B) : unió

Örököljünk a szülőtől, de zárjuk le a kijáratok felét:
V = inv(P * D[index]) * P;

Közbülső falak eltüntetése
V = uni(V, M[index]);

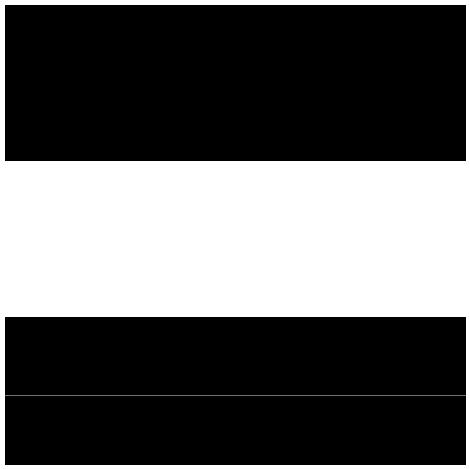
Cella felosztása

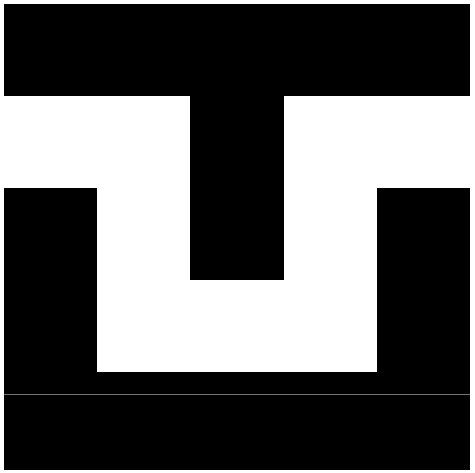
```
vec4 W1; // függőleges v. vízszintes
vec4 W2; // bal/lenn - jobb/fenn

if(R.x < 0.25) {  // right
  W1 = vec4(1,0,1,0);
  W2 = vec4(M[index].x == 1);
}
else if(R.x < 0.5){  // top
  W1 = vec4(0,1,0,1);
  W2 = vec4(M[index].y == 1);
}
else if(R.x < 0.75) {  // left
  W1 = vec4(1,0,1,0);
  W2 = vec4(M[index].z == 1);
}
else{  // bottom
  W1 = vec4(0,1,0,1);
  W2 = vec4(M[index].w == 1);
}

V = V * uni(uni(W1, W2), inv(M[index]));
```

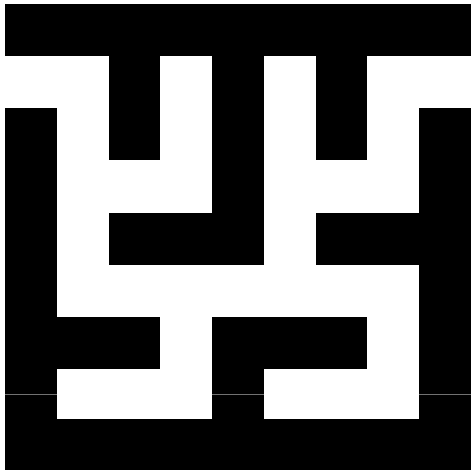
Fontos, hogy a négy cella ugyanazt a döntést hozza a felosztásról. Az *R* véletlen számot az előző szintről olvassuk:

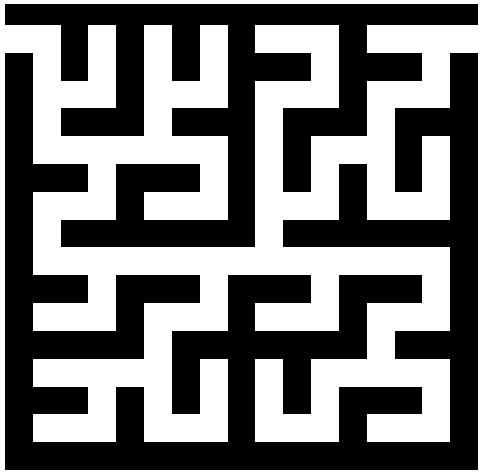
```
vec4 R = textureLod(inputTex2,
                    fTexCoord, level + 1);
```

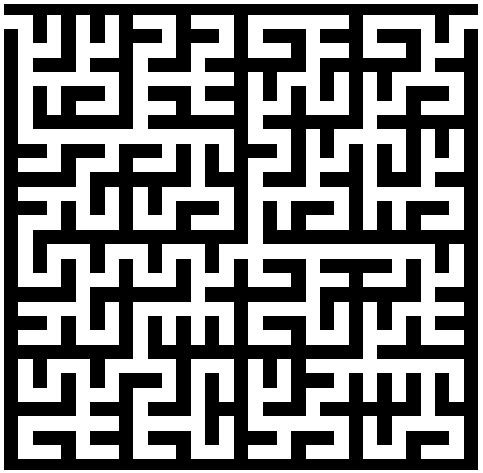
Labirintus felépítése

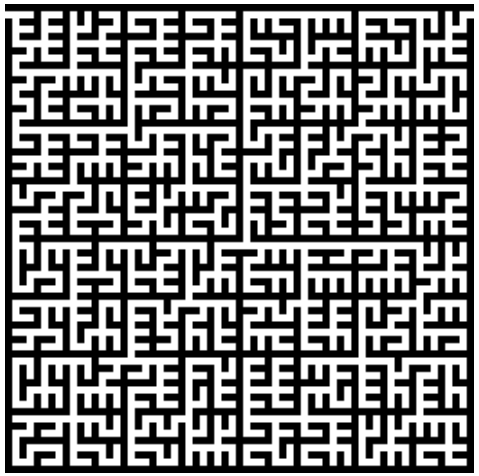










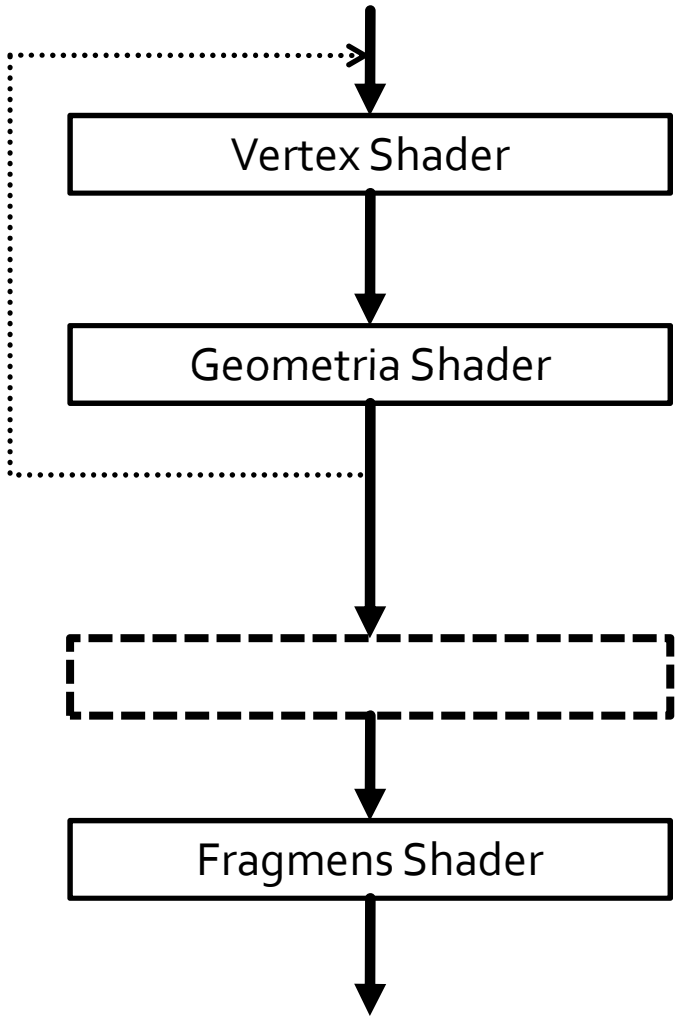


Geometria shader

- Opcionális lépcső
- Primitíveken dolgozik

- Bemenet: egy primitív
- Kimenet: egy vagy több

- A shader kimenete visszaköthető



Geometria shader

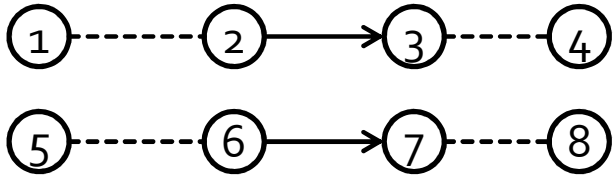
- Bementi primitívek

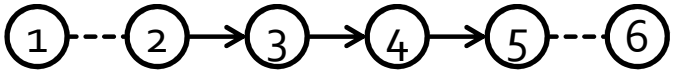
```
glProgramParameteri(shader, GL_GEOMETRY_INPUT_TYPE, tipus);
```

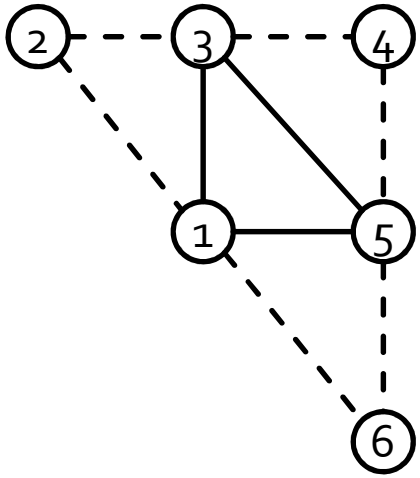
  - Pont
    - GL_POINTS
  - Szakasz
    - GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
  - Háromszög
    - GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
  - Adjacencia információ

Geometria shader

- Adjacencia
  - Szakasz
    - GL_LINES_ADJACENCY
    - GL_LINE_STRIP_ADJACENCY
  - Háromszög
    - GL_TRIANGLES_ADJACENCY
    - GL_TRIANGLE_STRIP_ADJACENCY







Geometria shader

- Kimeneti primitívek

```
glProgramParameteri(shader, GL_GEOMETRY_OUTPUT_TYPE, tipus);
glProgramParameteri(shader, GL_GEOMETRY_VERTICES_OUT, darab);
```

  - Pont
    - GL_POINTS
  - Szakasz
    - GL_LINE_STRIP
  - Háromszög
    - GL_LINE_STRIP

Geometria shader

- Speciális bemeneti változók
  - gl_ClipDistance[] : vágási információk
  - gl_PointSize[] : vertex méret a vertex shaderből
  - gl_Position : vertex pozíció
  - gl_PrimitiveIDIn : a feldolgozott primitív sorszáma
- Speciális kimeneti változók
  - A bemeneti változók
  - gl_Layer : melyik rétegbe tegye a fragmens shader

    (pl. cube map rendereléshez)

Geometria shader

- Primitívek generálása
  - Vertex információk beállítása
  - Vertex lezárása

```
EmitVertex();
```

  - Primitív lezárása

```
EndPrimitive();
```

Geometria shader

- Példa

```
#version 130
#extension GL_EXT_geometry_shader4 : enable

in vec2 vTexCoord[];
out vec2 fTexCoord;

void main(void){
 for(int i=0; i < gl_VerticesIn; ++i){
   gl_Position = gl_PositionIn[i];
   fTexCoord = vTexCoord[i];
   EmitVertex();
 }
 EndPrimitive();

 for(int i=0; i < gl_VerticesIn; ++i){
   gl_Position = gl_PositionIn[i].yxzw;
   fTexCoord = vTexCoord[i].yx;
   EmitVertex();
 }
 EndPrimitive();
}
```

Geometria shader

- Primitívek újrafeldolgozása
  - Transform feedback

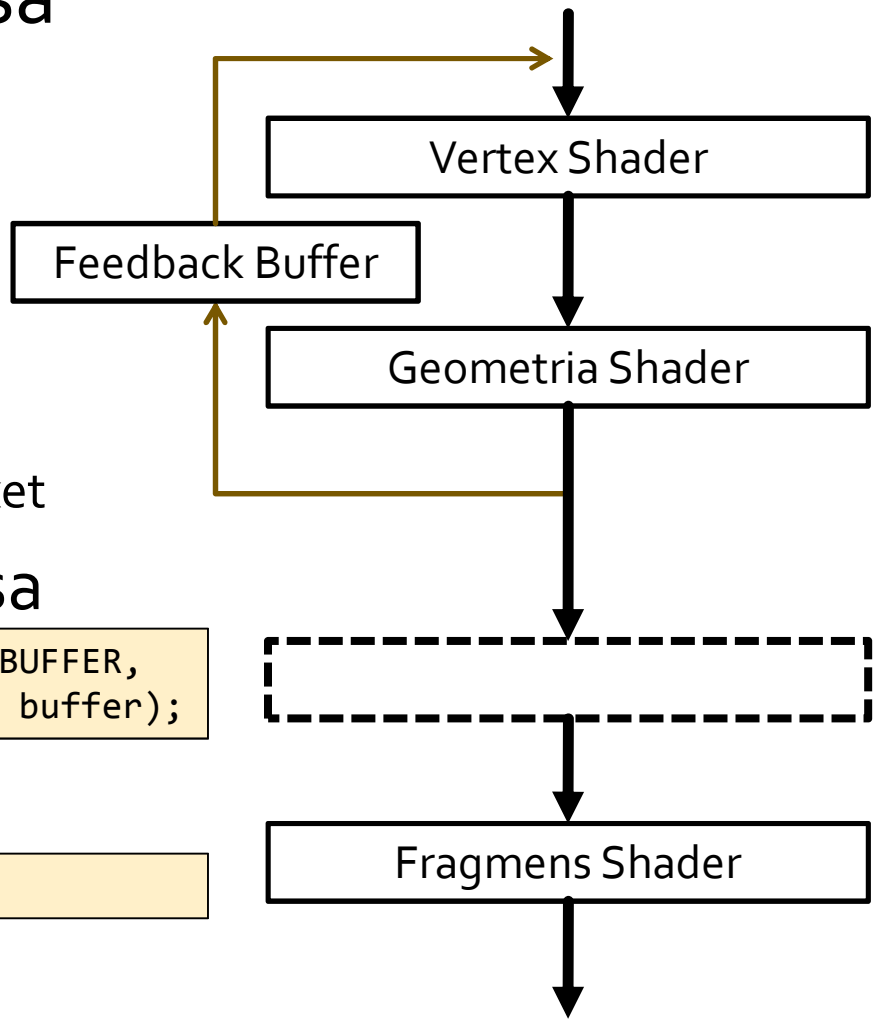
```
glBeginTransformFeedback(mode);
// glDrawArrays(...);
glEndTransformFeedback();
```

    - Feedback mód
      - Megadja a használható primitíveket
  - Feedback buffer kiválasztása

```
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,
                                    index, buffer);
```

  - Tulajdonságok kiválasztása

```
glTransformFeedbackVaryings(...);
```



Geometria shader

- Információ a geometria shader működéséről
  - Primitive query

```
GLuint outputQuery;
glGenQueries(1, &outputQuery);
glBeginQuery(mode, outputQuery);

...

glEndQuery(mode);
```

  - Query mód
    - GL_PRIMITIVES_GENERATED
      - Mennyi primitívet állított elő a geometria shader
    - GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN
      - Mennyi primitívet tudott a feedback bufferbe írni a shader

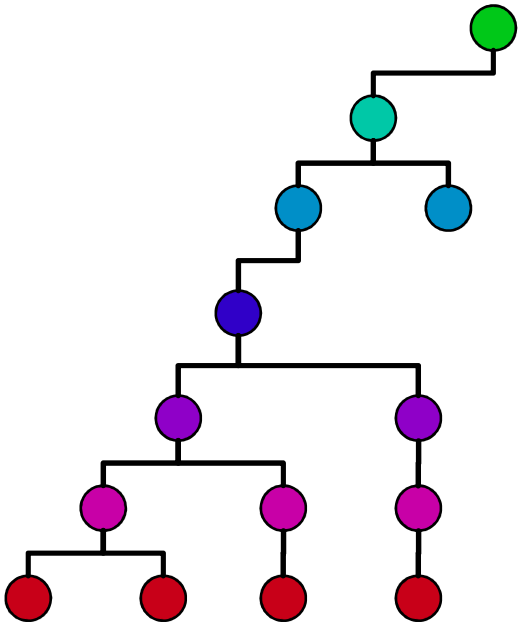
Geometria shader

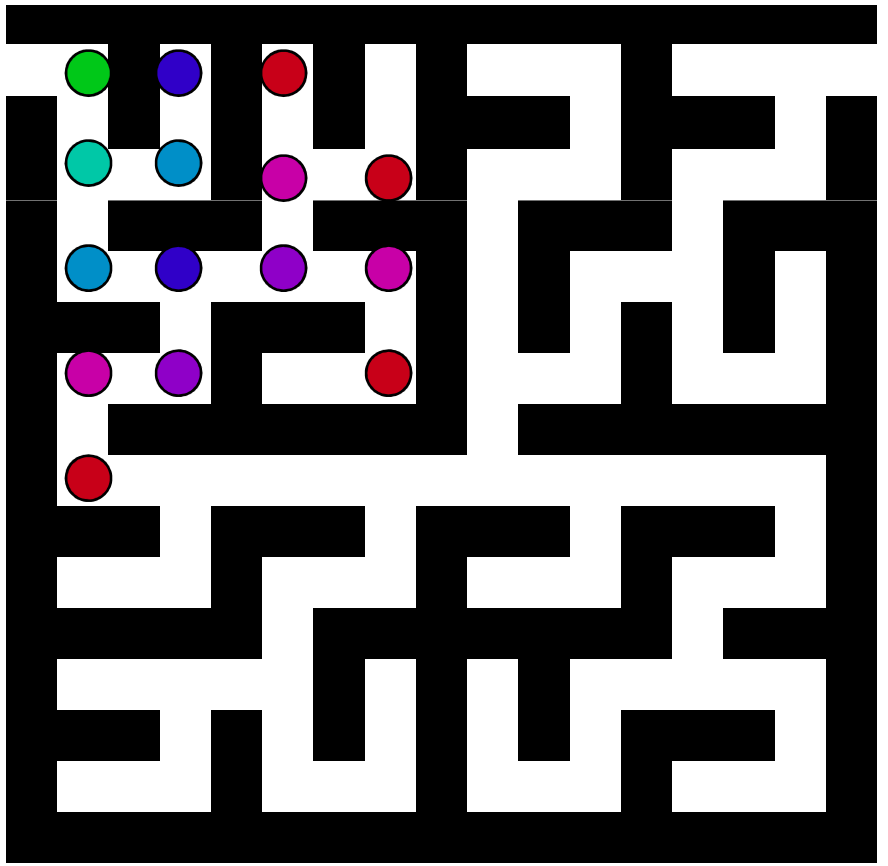
- Információ a geometria shader működéséről
  - A Query eredményének lekérdezése

```
GLuint outPointCount = 0;
GLuint succeded = 0;

while(!succeded){
  glGetQueryObjectiv(outputQuery, GL_QUERY_RESULT_AVAILABLE, &succeded);
}

glGetQueryObjectiv(outputQuery, GL_QUERY_RESULT, &outPointCount);
```

Labirintus bejárása

- Elágazó folyamat, szélességi gráf bejárás





Legrövidebb út, fő lépések

- A bejárás során egy cellának egy pont primitív fog megfelelni
  - V(x,y,d,n)
    - x,y: cella index
    - d: eddig megtalált legrövidebb úthossz
    - n: a legrövidebb úton melyik szomszédból jutottunk ide (irány index)
- A legrövidebb úthosszat egy textúrában tároljuk
  - Textúra inicializálása (0,0,0,0)
- A start cellából indulva azonosítjuk a szomszédjait
- Ha az adott szomszédba vezet út, megvizsgáljuk, hogy az ahhoz tartozó eddig megtalált legrövidebb úthossz hosszabb-e a csomópontunkhoz tartozó (d + 1) értéknél (vagy még nem is jártunk ebben a cellában)
- Ha igen, a cella értékét frissíteni kell, és szomszédait meg kell látogatni (előző lépés ismétlése)

Legrövidebb út problémák

- Elágazás: egy pontból több pont lesz
  - megoldás: Geometry Shader
- Rekurzió: a szomszédokra újra végre kell hajtani a feladatot
  - megoldás: Vertex Transform Feedback

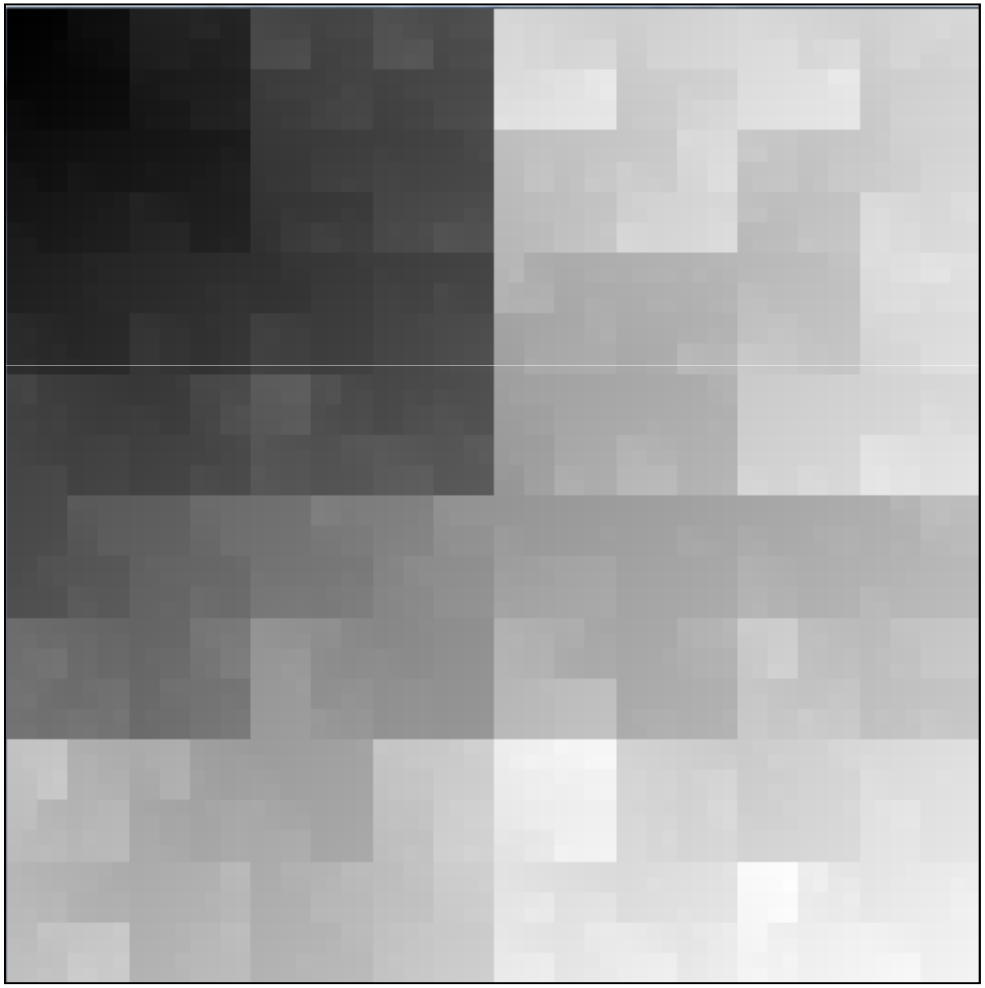
Legrövidebb út keresés

```
...

vec2 windowSize = textureSize(inputTex2, 0);
vec4 vertexdata = gl_PositionIn[0];

//read neighbour information
vec4 neighInf = texelFetch(inputTex1, ivec2(vertexdata.xy), 0);
vec4 storedCelldata = texelFetch(inputTex2, ivec2(vertexdata.xy), 0);

if(storedCelldata.z > vertexdata.z || storedCelldata.w == 0){
  celldata = vertexdata;
  vec2 wCoord = vertexdata.xy / windowSize * 2.0 - 1.0;
  gl_Position = vec4(wCoord,0,1);
  EmitVertex();
}

...
```

Legrövidebb út keresés

```
//emit neighbours if connected

//right
if(neighInf.r == 1){
  celldata = vertexdata;
  celldata.x += 1;
  celldata.z += 1;
  celldata.w = 3;
  storedCelldata = texelFetch(inputTex2, ivec2(celldata.xy), 0);
  if((storedCelldata.z > celldata.z || storedCelldata.w == 0) &&
                                        celldata.x <   windowSize.x){
    vec2 wCoord = celldata.xy / windowSize * 2.0 - 1.0;
    gl_Position = vec4(wCoord,0,1);
    EmitVertex();
    }
}
…
EndPrimitive();
```

Legrövidebb út hossza



Legrövidebb út egy célcellába, bejárás

- Hasonló a feladat, csak nincs elágazás
- Célcellából indulunk (pontprimitív)
- Kiolvassuk, hogy melyik cellából jutottunk ide
- Ez a cella lesz az új pont primitív (ki is rajzoljuk a képernyőre, hogy lássuk)
- Addig folytatjuk, míg el nem jutunk a start cellába (ahol az úthossz 0)

Labirintus bejárása

```
#version 130
#extension GL_EXT_geometry_shader4 : enable

uniform sampler2D inputTex1; //cell information
out vec4 celldata;

void main(void){
  vec2 windowSize = textureSize(inputTex1, 0);
  vec4 vertexdata = gl_PositionIn[0];

  vec4 storedCelldata = texelFetch(inputTex1, ivec2(vertexdata.xy), 0);

  if(storedCelldata.z == 0) return;

  if(storedCelldata.w == 1){ // right
    celldata = vertexdata;
    celldata.x += 1;
    vec2 wCoord = (celldata.xy * 2 + 1) / (windowSize * 2 + 1) * 2.0 - 1.0;
    gl_Position = vec4(wCoord,0,1);
    EmitVertex();
  }
  ...
  EndPrimitive();
}
```

Kijutottunk a labirintusból!!!!!!!