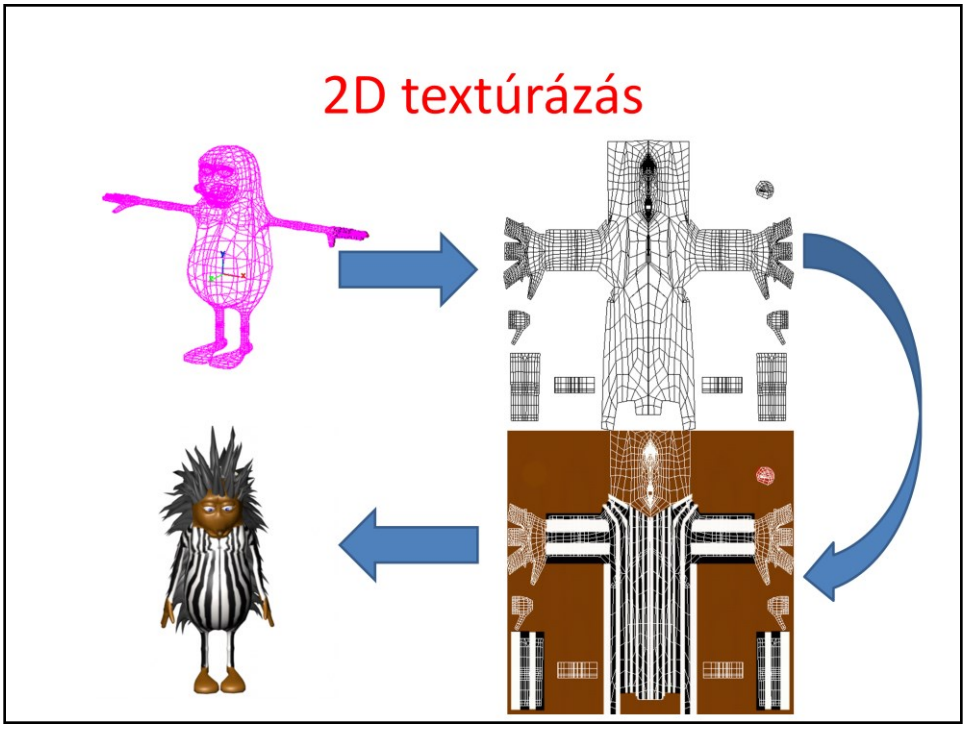
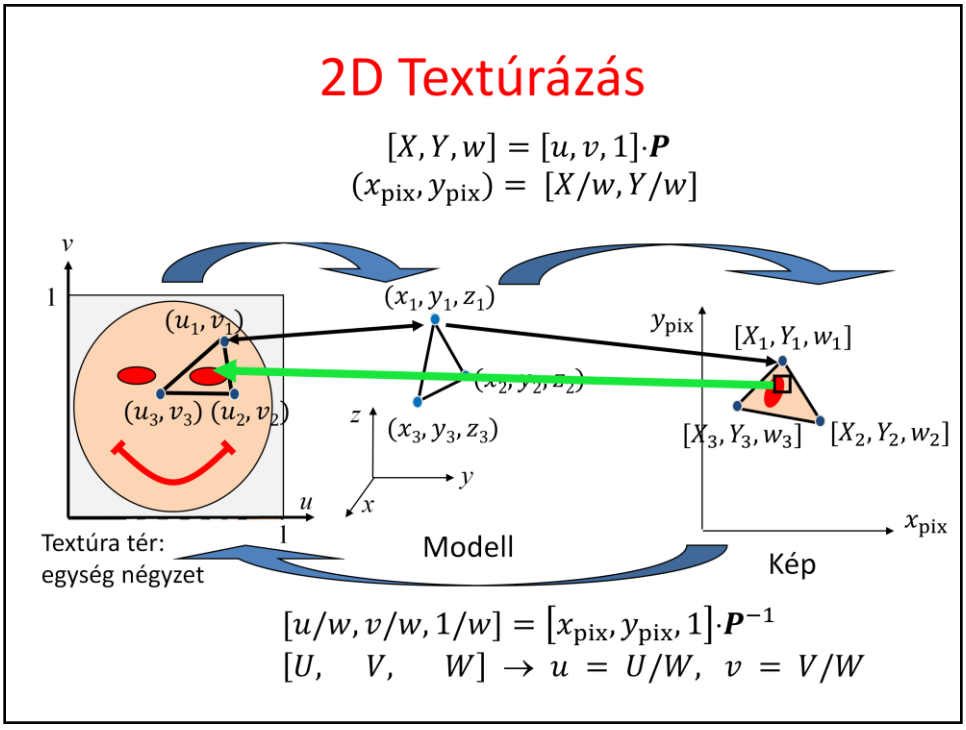
“Everything must be made as simple as possible. But not simpler.”
Albert Einstein

2D textúrázás

Szirmay-Kalos László


2D textúrázás

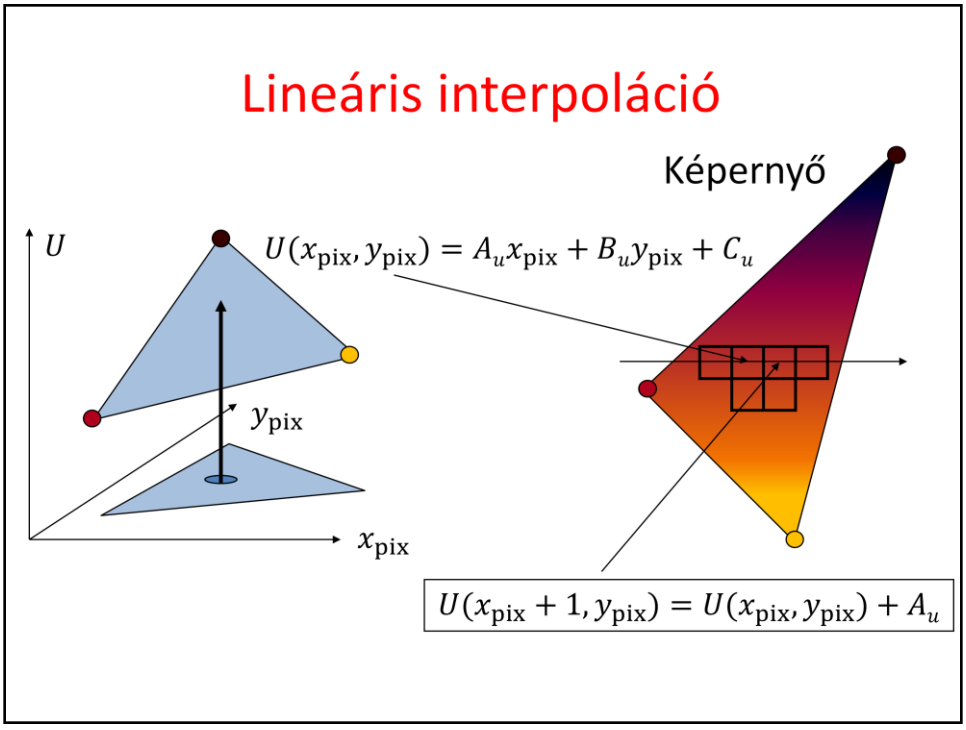


2D texture mapping can be imagined as wallpapering. We have a wallpaper that defines the image or the function of a given material property. This image is defined in texture space as a unit rectangle. The wallpapering process will cover the region with this image. This usually implies the distortion of the texture image. To execute texturing, we have to find a correspondence between the region and the 2D texture space. After vectorization, the region is a triangle mesh, so for each triangle, we have to identify a 2D texture space triangle, which will be painted onto the model triangle. The definition of this correspondence is called **parameterization.** A triangle can be parameterized with an affine transformation (x, y are linear functions of u, v). This can be seen by counting the number of unknowns in a pair of linear functions (6) and counting the number of constraints for an affine transformation to map one given triangle onto another given triangle (this is also 6). So for unknown a_x, b_x, c_x, a_y, b_y, c_y parameters, we can establish a system of equations where the number of unknowns is the same as the number of equations.

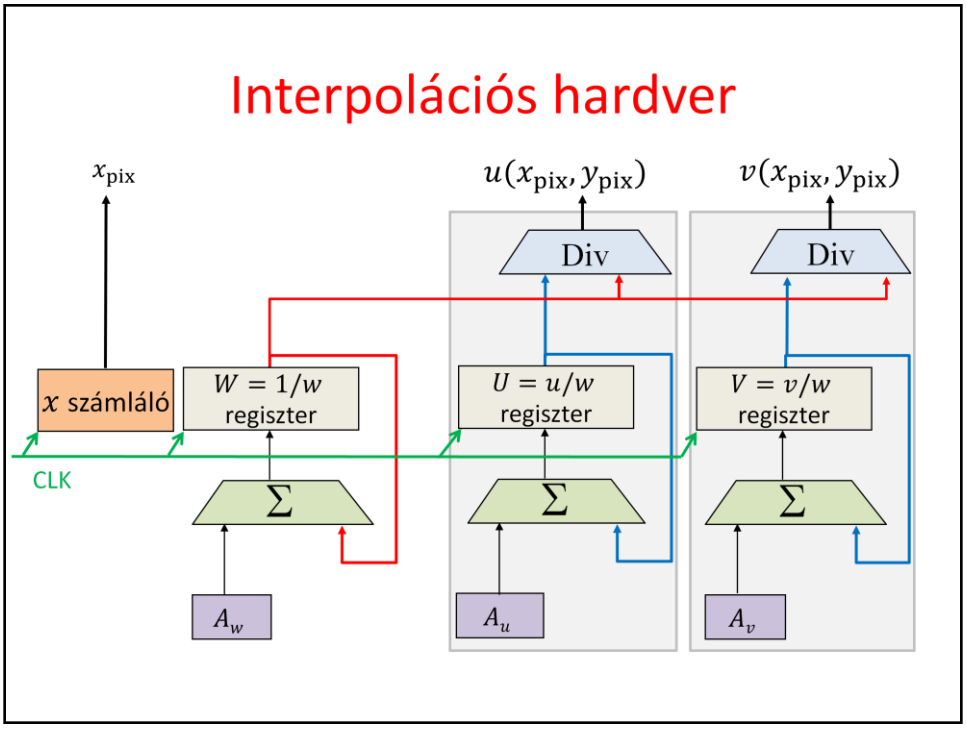
Screen space coordinates (X,Y) are obtained with affine or to be general with homogeneous linear transformation from x, y. Thus the mapping between texture space and screen space is also a homogeneous linear transformation:

The triangle is rasterized in screen space. When a pixel is processed, texture

coordinate pair u, v must be determined from pixel coordinates X,Y, which requires the execution of the inverse of this homogeneous linear transformation. If we need the result in Cartesian coordinates, homogeneous division is also needed. In other words, homogeneous coordinates [*U, V, H*] depend linearly on pixel coordinates X and Y, and then Cartesian texture coordinates are obtained as $u = U/H$, $v = V/H$.

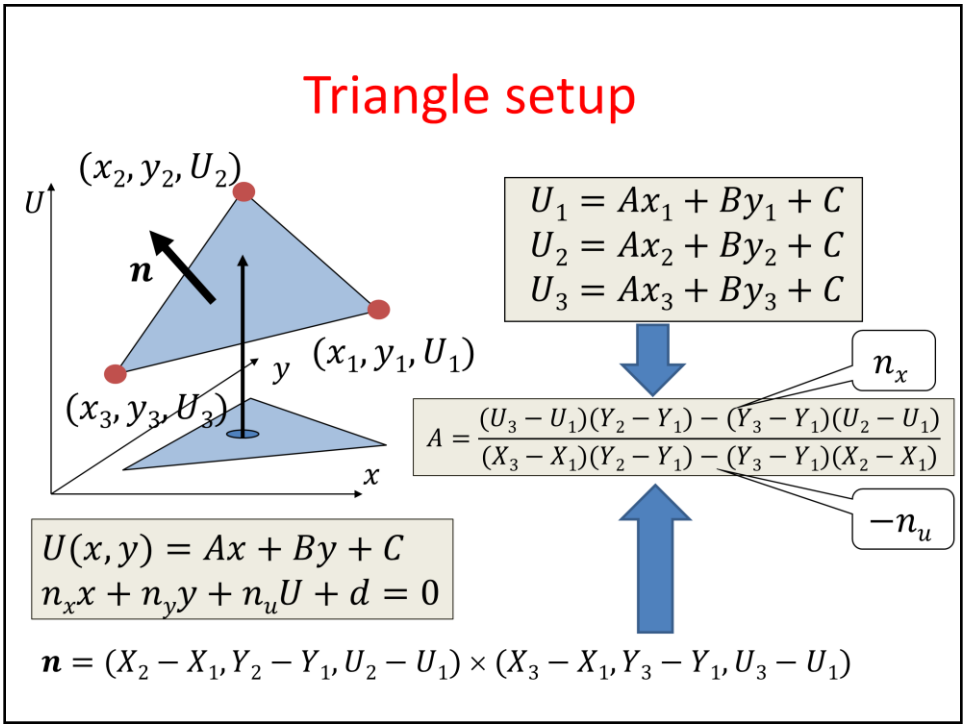


The crucial problem is then the linear interpolation of the color, properties in general, or texture coordinates for internal pixels. This interpolation must be fast as we have just a few nanoseconds for each pixel. Let us denote the interpolated property by U. Linear interpolation means that U is a linear function of pixel coordinates X, Y. Linear function $U(X,Y) = A_u X + B_u Y + C_u$ evaluation would require two multiplications and two additions. This can be reduced to a single addition if we use the incremental concept and focus on the difference between the U values of the current and previous pixels in this scanline: $U(X+1,Y) = U(X,Y) + A_u$.



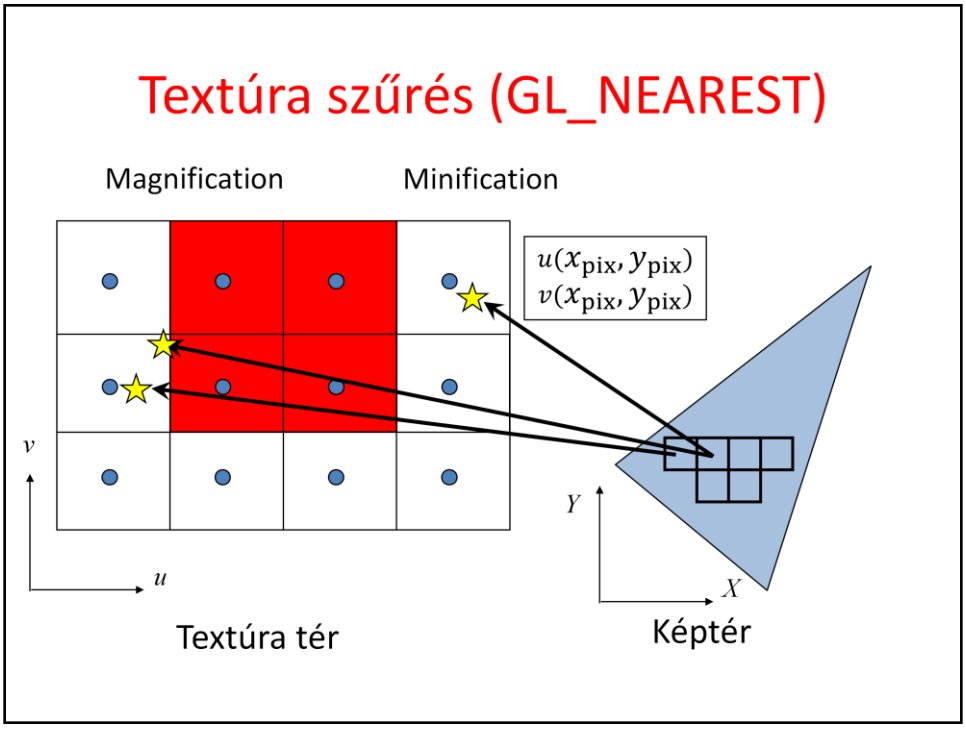
Such an incremental algorithm is easy to be implemented directly in hardware. A counter increments its value to generate coordinate X for every clock cycle. A register that stores the actual U coordinate in fixed point, non-integer format (note that increment *A* is usually not an integer). This register is updated with the sum of its previous value and *A* for every clock cycle. For a texture coordinate, we execute linear interpolation for the three homogeneous coordinates [*U, V, H*].

From these, the u, v Cartesian texture coordinates are obtained by two additional division.



The final problem is how increment *A* is calculated. One way of determining it is to satisfy the interpolation constraints at the three vertices. It means that substituting the X, Y coordinates of the vertices into the linear expression, we should obtain the value to be interpolated at the three vertices. This is a system of linear equations for unknown A, B, C.

The other way is based on the recognition that we work with the plane equation where X,Y,U coordinates are multiplied by the coordinates of the plane's normal. The normal vector, in turn, can be calculated as a cross product of the edge vectors.

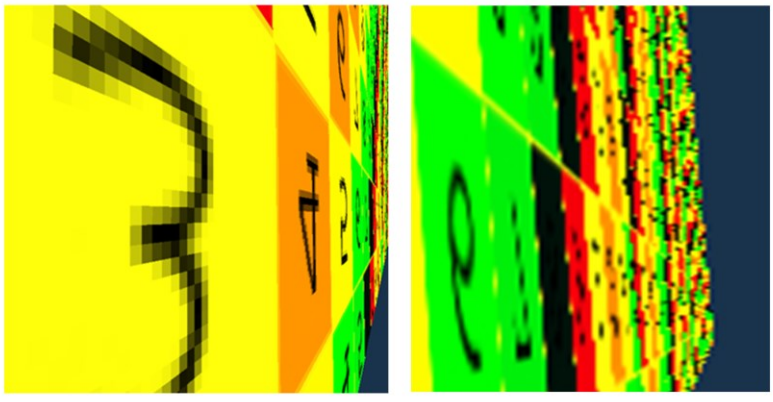


Rasterization visits pixels inside the projection of the triangle and maps the center of the pixel from screen space to texture space to look up the texture color. This mapping will result in a point that is in between the texel centers.

Suppose that we step onto the next pixel. Consequently, the transformed u, v texture coordinates also change. This change can be smaller than a texel or bigger than a texel. Looking at this situation from the other direction, the first case is when a texel covers multiple pixels. This is called **magnification**. The second case is when many texels are mapped to a single pixel and the pixel color is determined by that texel which is lucky enough to be mapped onto the pixel center. This case is called **minification.**

From signal processing point of view, the texture can be considered as a signal, which is sampled at the pixel centers. In the optimal case, one texel corresponds to one pixel. In case of magnification, texels will be large, which is oversampling. In case of minification, texels are too small and are undersampled and the result will be a mess or noise.

Textúratér és képtér kapcsolata

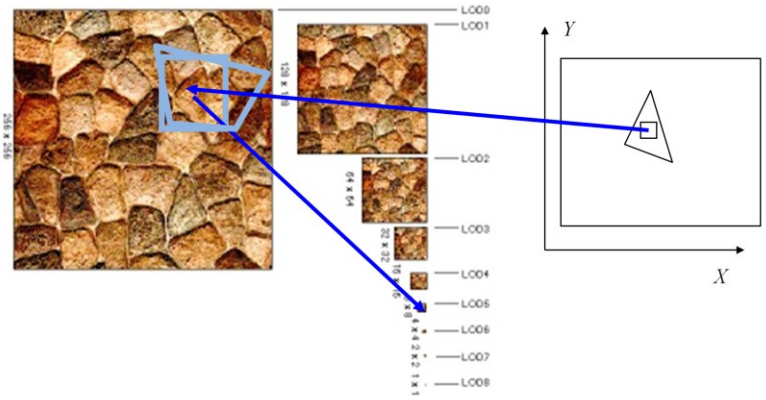


Magnification Minification

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

In case of oversampling or undersampling artifacts occur. Magnification makes the individual texels obvious. Minification converts the pattern to a noise. When these images were rendered, the pixel is colored with the texel covering its center. This sampling strategy is called **GL_NEAREST** in OpenGL. The sampling strategy can be set with the **glTexParameter** function separately for the minification (**GL_TEXTURE_MIN_FILTER**) and magnification (**GL_TEXTURE_MAG_FILTER**).

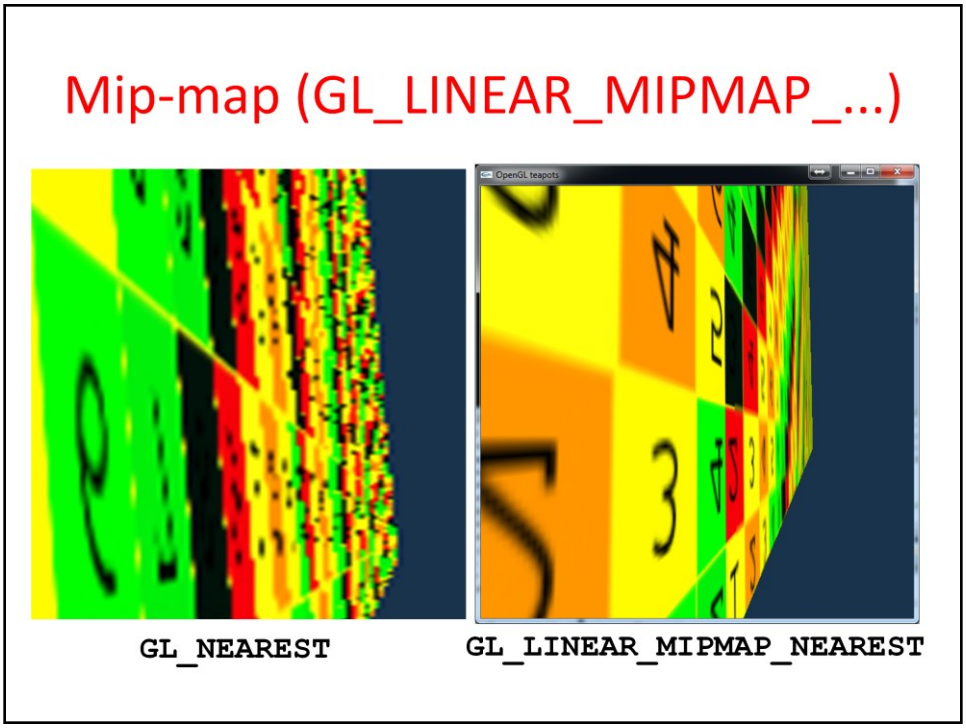
Mip-map (multum in parvo)
Minification-ra jó




```
a) glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR_MIPMAP_NEAREST); // Mip-mapping

b) glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR_MIPMAP_LINEAR); // Tri-linear filtering
```

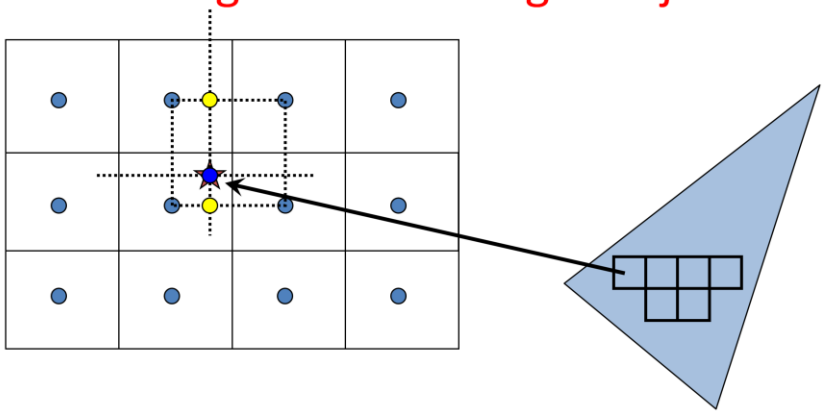
The solution for the undersampling problem is filtering. Instead of mapping just the center of the pixel, the complete pixel rectangle must be mapped to texture space at least approximately, and the average of texel colors in this region should be returned as a color. However, it would be too time consuming.

One efficient approximation is to prepare the texture not only in its original resolution, but also in half resolution, quarter resolution, etc. where a texel represents the average color of a square of the original texel. During rasterization, OpenGL estimates the magnification factor, and looks up the appropriate version of filtered, downsample texture. The collection of the original and downsampled textures is called mip-map.



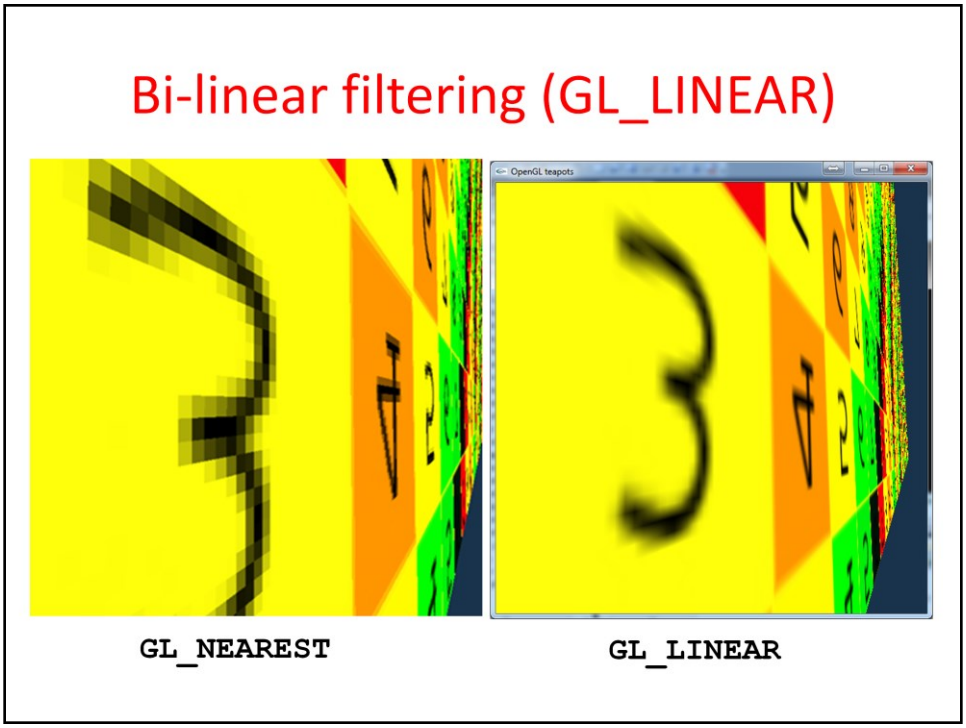
Indeed, with mip-mapping undersampling artifacts disappear. Mip-mapping can be selected with **GL_LINEAR_MIPMAP_NEAREST** texture sampling strategy.

Bi-linear textúra szűrés (GL_LINEAR)
Magnification-ra igazán jó



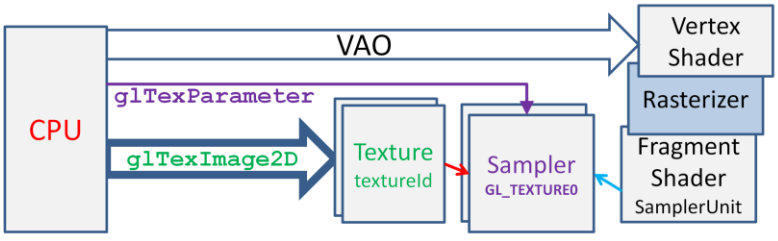
```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

There is another simpler filtering scheme, which is particularly good for magnification, i.e. to reduce oversampling artifacts. In this method, when a pixel center is mapped to texture space, not only the closest texel is obtained but the four closest ones, and the filtered color is computed as the bi-linear interpolation of their colors.



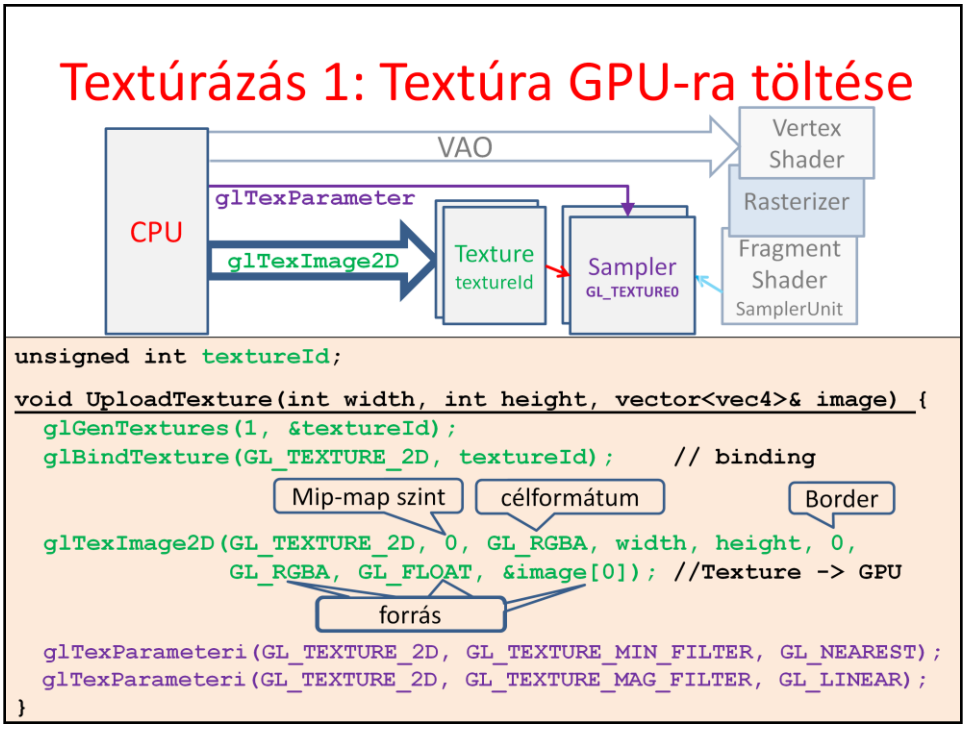
As these images demonstrate, bi-linear filtering smears the ugly texel edges.

Textúrázás a GPU-n



If we wish to use texturing, four steps should be executed. In this slide we show how to upload an image to the GPU that is to be used as a 2D texture. Additionally, texture sampling/filtering is also specified, which is neareast neighbor in case of minification and bi-linear filtering in case of magnification.

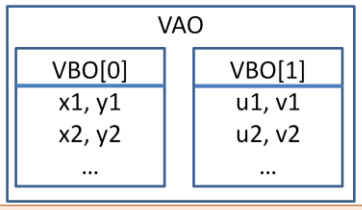
Note that OpenGL is an output library, so it gives no help to create or read a texture from a file. This is the programmers' responsibility, which should be done in the LoadImage function.



If we wish to use texturing, four steps should be executed. In this slide we show how to upload an image to the GPU that is to be used as a 2D texture. Additionally, texture sampling/filtering is also specified, which is neareast neighbor in case of minification and bi-linear filtering in case of magnification.

Note that OpenGL is an output library, so it gives no help to create or read a texture from a file. This is the programmers' responsibility, which should be done in the LoadImage function.

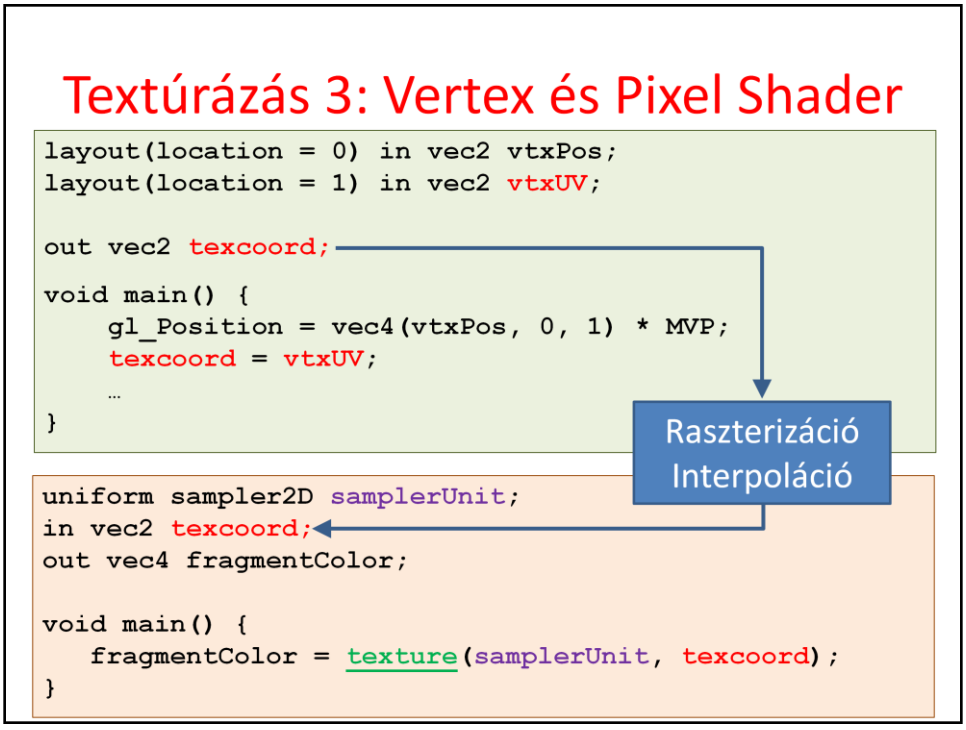
Textúrázás 2: Objektumok felszerelése textúra koordinátákkal




```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(2, vbo);// Generate 2 vertex buffer objects

// vertex coordinates: vbo[0] -> Attrib Array 0 -> vertices
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
float vtxs[] = {x1, y1, x2, y2, …};
glBufferData(GL_ARRAY_BUFFER, sizeof(vtxs),vtxs, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);

// vertex coordinates: vbo[1] -> Attrib Array 1 -> uvs
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
float uvs[] = {u1, v1, u2, v2, …};
glBufferData(GL_ARRAY_BUFFER, sizeof(uvs), uvs, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, NULL);
```

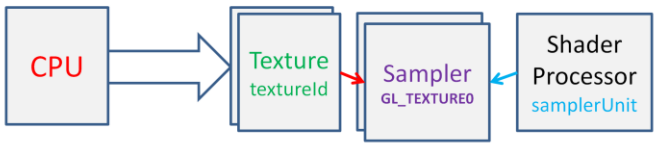
The second step equips the object to be textured with texture coordinates or so called uvs. That is, for every vertex we also specify a texture space point from where the color or data should be fetched if this point is rendered.

In the shown program or VAO has two VBOs, one stores the modeling space Cartesian coordinates of the points and the second the texture space coordinates of the same points. We direct modeling space coordinates to input register 0 and texture space coordinates to input register 1.



The vertex shader gets the location of the point and also the uv associated with this point. The location is transformed to normalized device space, the texture coordinate is only copied by the vertex shader. The fixed function hardware interpolates the texture coordinates and the fragment shader looks up the texture memory via the sampler unit.

Textúrázás 4: Aktív textúra és sampler




```
unsigned int textureId;

void Draw( ) {
  int sampler = 0; // which sampler unit should be used

  int location = glGetUniformLocation(shaderProg, "samplerUnit");
  glUniform1i(location, sampler);

  glActiveTexture(GL_TEXTURE0 + sampler);  // = GL_TEXTURE0
  glBindTexture(GL_TEXTURE_2D, textureId);

  glBindVertexArray(vao);
  glDrawArrays(GL_TRIANGLES, 0, nVtx);
}
```

When the object is rendered, we should select the active texture and also connect the shader processor to this texture via a sampler unit. The samplerUnit variable of the shader program gets the id of the sampler unit (zero in this example). With the glActiveTexture and glBindTexture calls, we also establish the connection between the texture in the texture memory and the sampler unit.

2D textúrázás
1. Program: Textúrázott négyszög

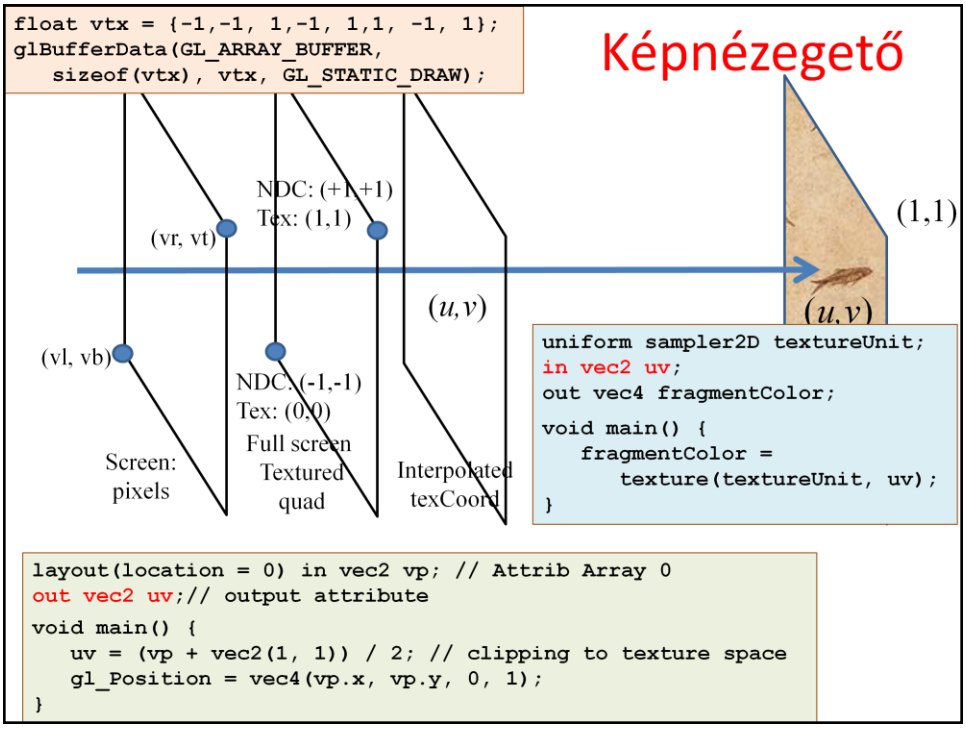
Szirmay-Kalos László

"The message of this lecture is that black holes ain't as black as they are painted. So if you feel you are in a black hole, don't give up – there's a way out."

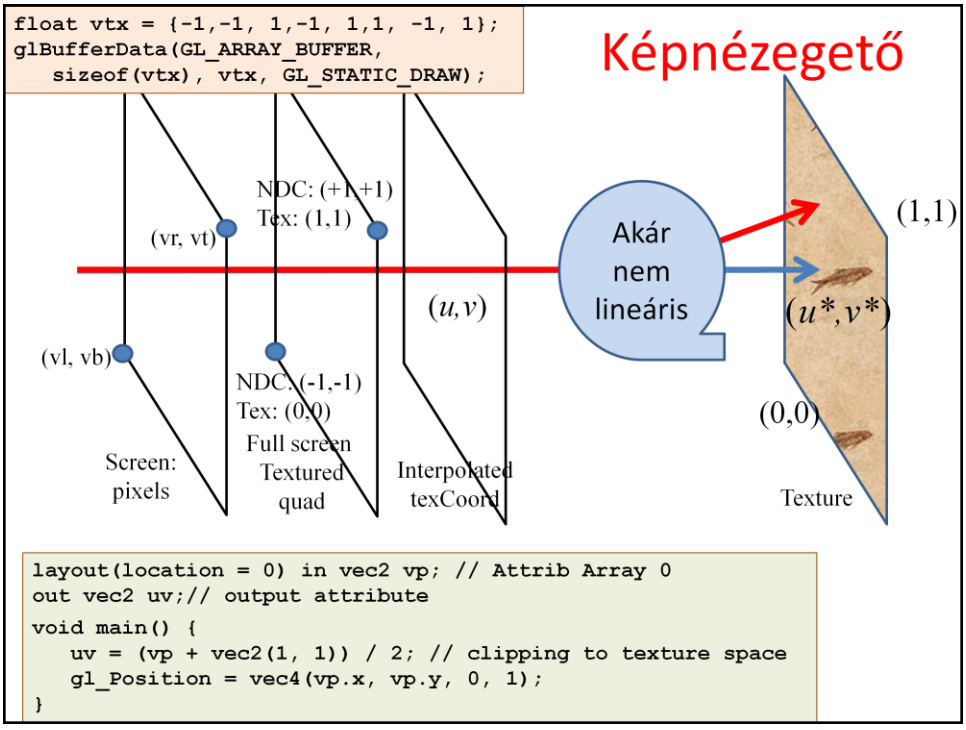
Stephen Hawking

Képnézegető és nemlineáris 2D képeffektusok

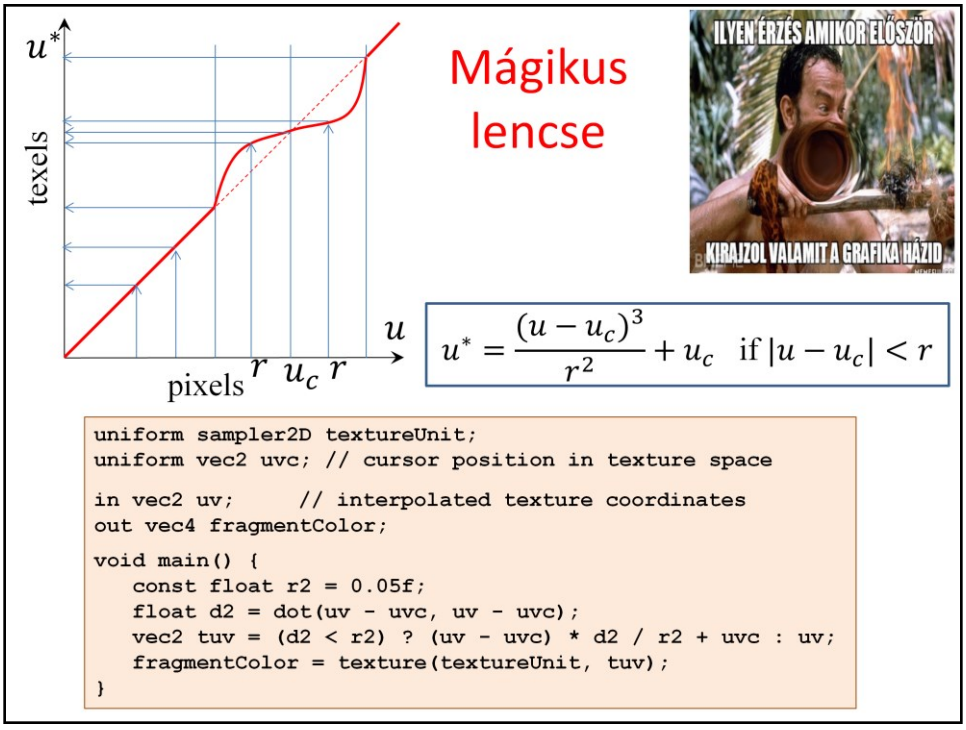
Szirmay-Kalos László



Let us implement an image viewer program. To copy an image into the raster memory, we should draw a full viewport sized quad textured with the image. In normalized device space, the full viewport quad has corners (-1,-1), (1,-1), (1,1), (-1,1), which are stored in the vbo. From these, the texture coordinates (0,0), (1,0), (1,1), (0,1) addressing the four corners of the texture rectangle are computed by the vertex shader. The fragment shader looks up the texture with the interpolated uv.



If we modify the interpolated texture coordinate by some function, exciting image effects can be produced.




```
uniform sampler2D textureUnit;
uniform vec2 uvc; // cursor position in texture space

in vec2 uv;     // interpolated texture coordinates
out vec4 fragmentColor;

void main() {
   const float r2 = 0.05f;
   float d2 = dot(uv - uvc, uv - uvc);
   vec2 tuv = (d2 < r2) ? (uv - uvc) * d2 / r2 + uvc : uv;
   fragmentColor = texture(textureUnit, tuv);
}
```

Normally, the texture is fetched at interpolated texture coordinate uv, but now we shall transform it to tuv and get the texture at tuv. The transformation uv to tuv uses the texture space location of the cursor to make the effect interactive.

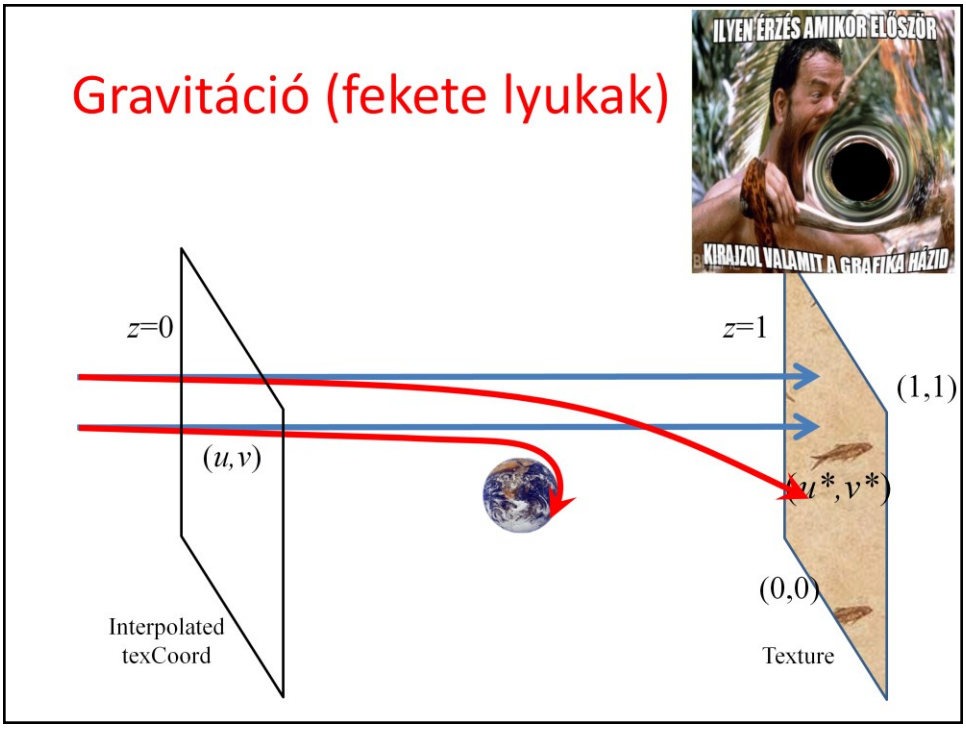
Note that if tuv is the translation of uv, then the effect corresponds to the translation of the image into the opposite direction. Similarly, if tuv is the rotated version of uv, we observe the effect as rotating the image backwards. Generally, the inverse of the transformation of uv to tuv will be applied to the image.

Note also that we can use non-linear transformations as well since finite number of texel centers are transformed. In this demo we modify the originally linear dependence of tuv (or u*) on uv and insert a non-linear cubic segment. Close to uc, the slope of the function is less than 1, i.e. when we step to the next pixel, the corresponding movement in the texture is less than normal. Here a texel is seen on more pixels, thus the image is magnified. When the slope is greater than 1, the image is contracted. So the seen effect is what a lens would produce.

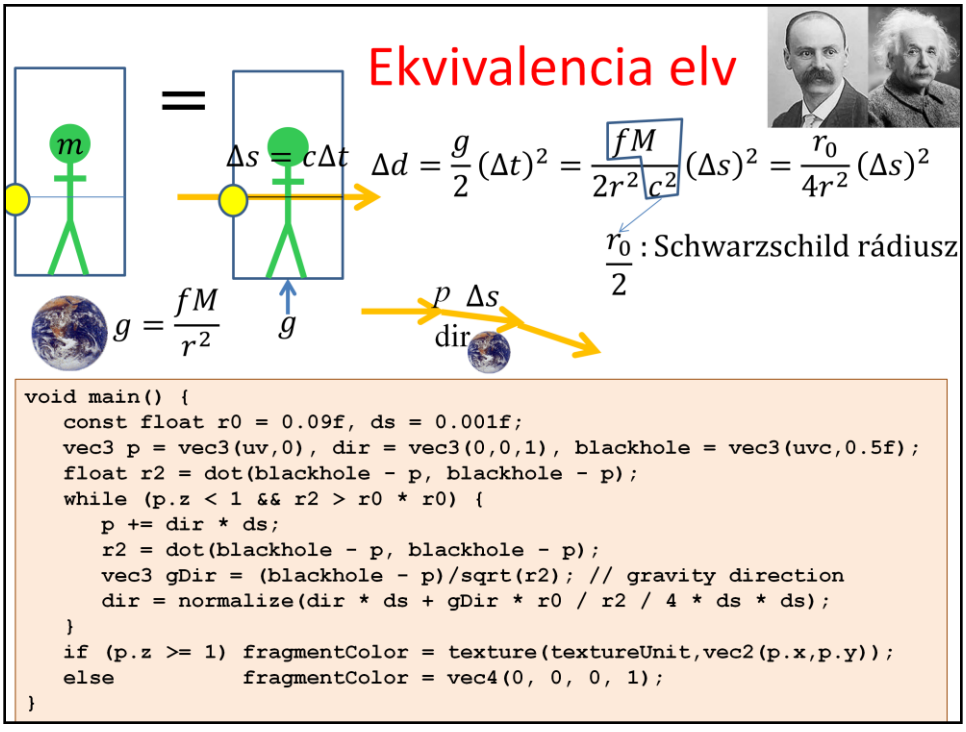
Örvény: Swirl


```
uniform sampler2D textureUnit;
uniform vec2 uvc; // cursor position in texture space

in vec2 uv;     // interpolated texture coordinates
out vec4 fragmentColor;

void main() {
   const float a = 8, alpha = 15;
   float ang = a * exp( -alpha * length(uv - uvc) );

   mat2 rotMat = mat2( cos(ang), sin(ang),
                      -sin(ang), cos(ang) );

   vec2 tuv = (uv - uvc) * rotMat + uvc;
   fragmentColor = texture(textureUnit, tuv);
}
```

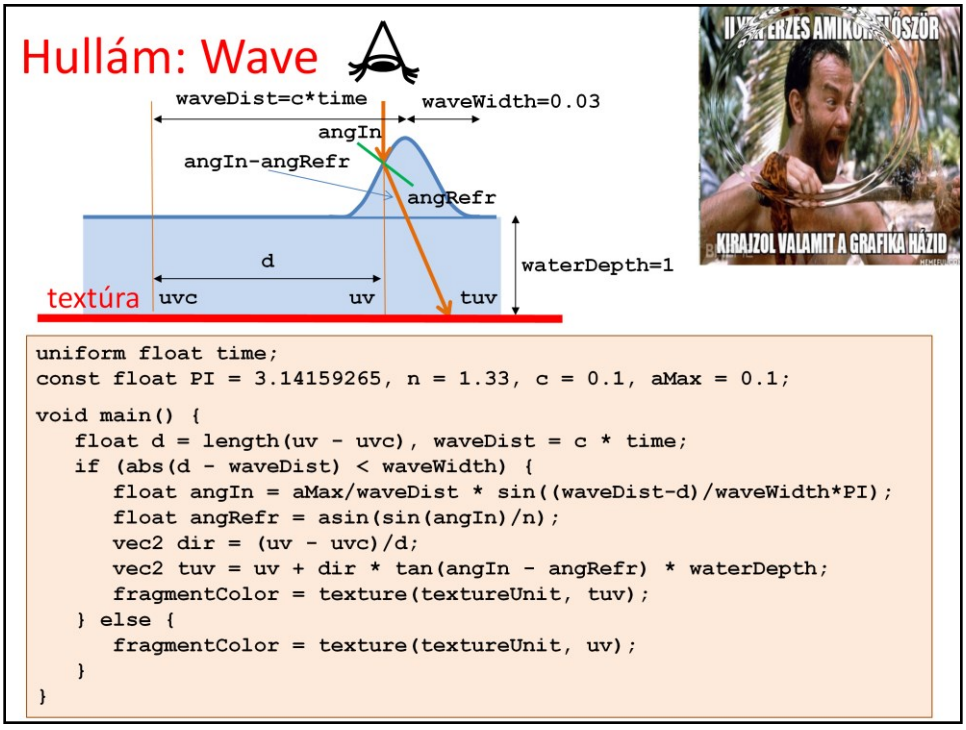
A swirl is a rotation where the angle (or speed) of the rotation grows towards the center of the swirl. A rotation is a mat2 matrix multiplication where the pivot point is the current cursor location, uvc. To control the angle of the rotation, we compute the distance from the center and use the a*exp(-alpha * x) function to obtain smaller angles farther from the center.



In this effect, we assume that a large mass but small size object, e.g. a black hole is in front of the image, which distorts space and also lines of sight.



The amount of distortion can be computed exploiting the equivalence principle that states that gravity and acceleration are the same and cannot be distinguished. We can easily determine the amount of light bending when our room is accelerating in a direction. Then, the same formula is applied when the room stands still but a black hole distorts the space.




```
uniform float time;
const float PI = 3.14159265, n = 1.33, c = 0.1, aMax = 0.1;

void main() {
   float d = length(uv - uvc), waveDist = c * time;
   if (abs(d - waveDist) < waveWidth) {
      float angIn = aMax/waveDist * sin((waveDist-d)/waveWidth*PI);
      float angRefr = asin(sin(angIn)/n);
      vec2 dir = (uv - uvc)/d;
      vec2 tuv = uv + dir * tan(angIn - angRefr) * waterDepth;
      fragmentColor = texture(textureUnit, tuv);
   } else {
      fragmentColor = texture(textureUnit, uv);
   }
}
```

In this final demo, we assume that image is covered by water and its surface is distorted by a wave started in a single point and moving in all directions with speed c. The width of the wave is waveWidth, its amplitude is decreased with the travelled distance. The distortion is computed by applying the Snells refraction law on the water surface.