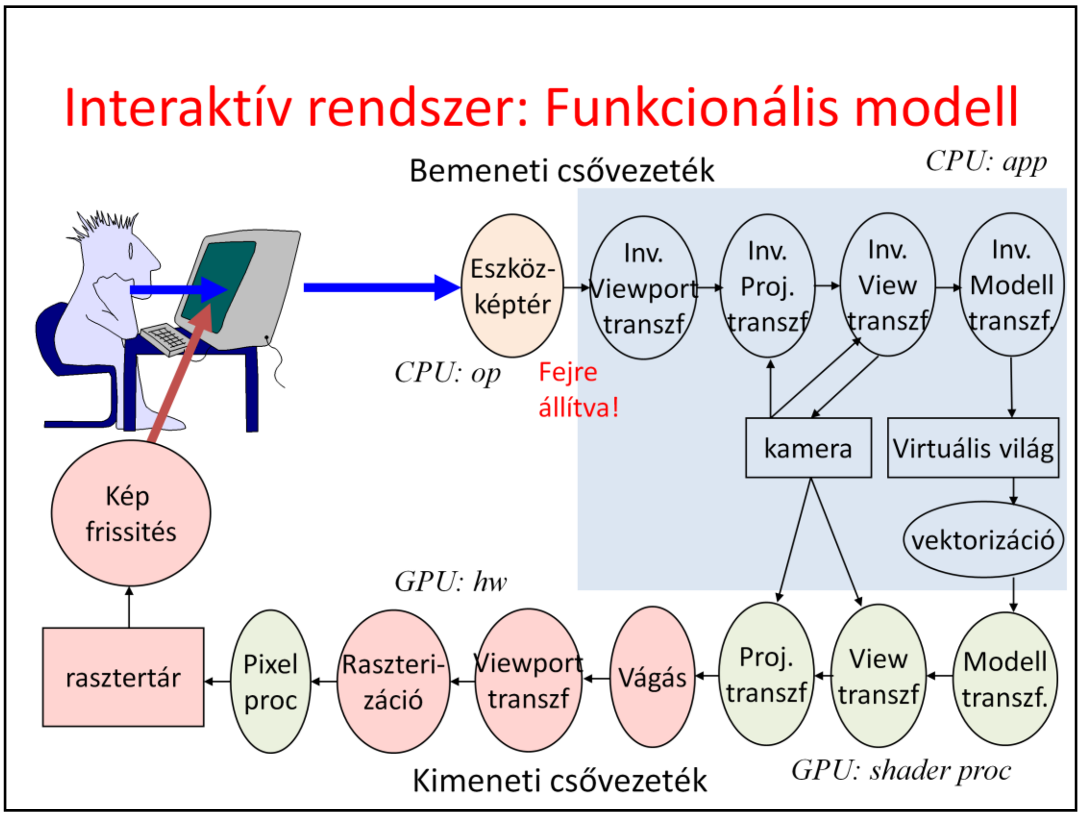
Grafikus hardver/szoftver alapok

Szirmay-Kalos László

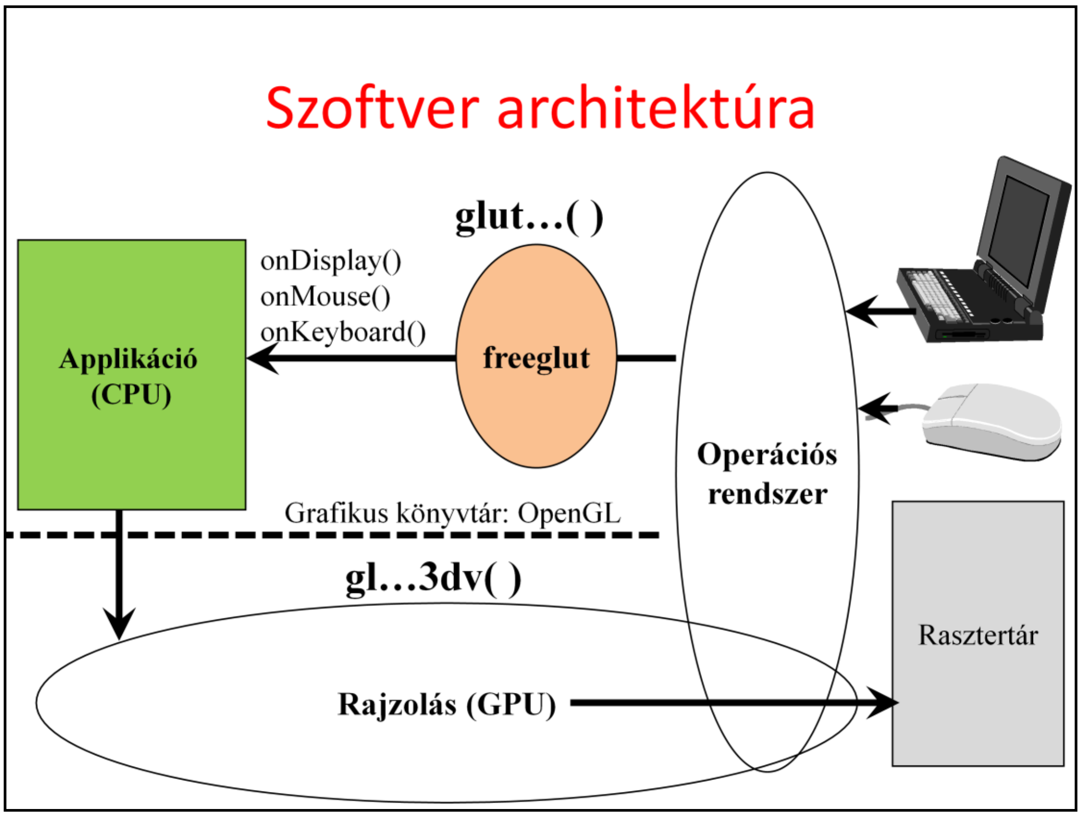


Interaktív rendszer: Funkcionális modell

From system point of view, a graphics application handles the user input, changes the internal state, called the virtual world by modeling or animating it, and then immediately renders the updated model presenting the image to the user. This way, the user immerses into the virtual world, i.e. he feels that he is promptly informed about its current state. The process from the input to the virtual world is called the input pipeline. Similarly, the process mapping the virtual world to the screen is the output pipeline.

The complete system is a (control) loop with two important points, the virtual world and the user. In the output pipeline, the virtual world is vectorized first since only lines and polygons can be transformed with homogeneous linear transformations. Then modeling, view and projection transformations are executed moving the current object to normalized device space. Here clipping is done, then the object is transformed to the screen, where it is rasterized. Before being written in the frame buffer, pixels can undergo pixel operations, needed, for example, to handle transparent colors. The frame buffer is read periodically to refresh the screen. The user can see the screen and interact with the content by moving the cursor with input devices and starting actions like pressing a button. Such actions generate events taking also the screen space position with them.

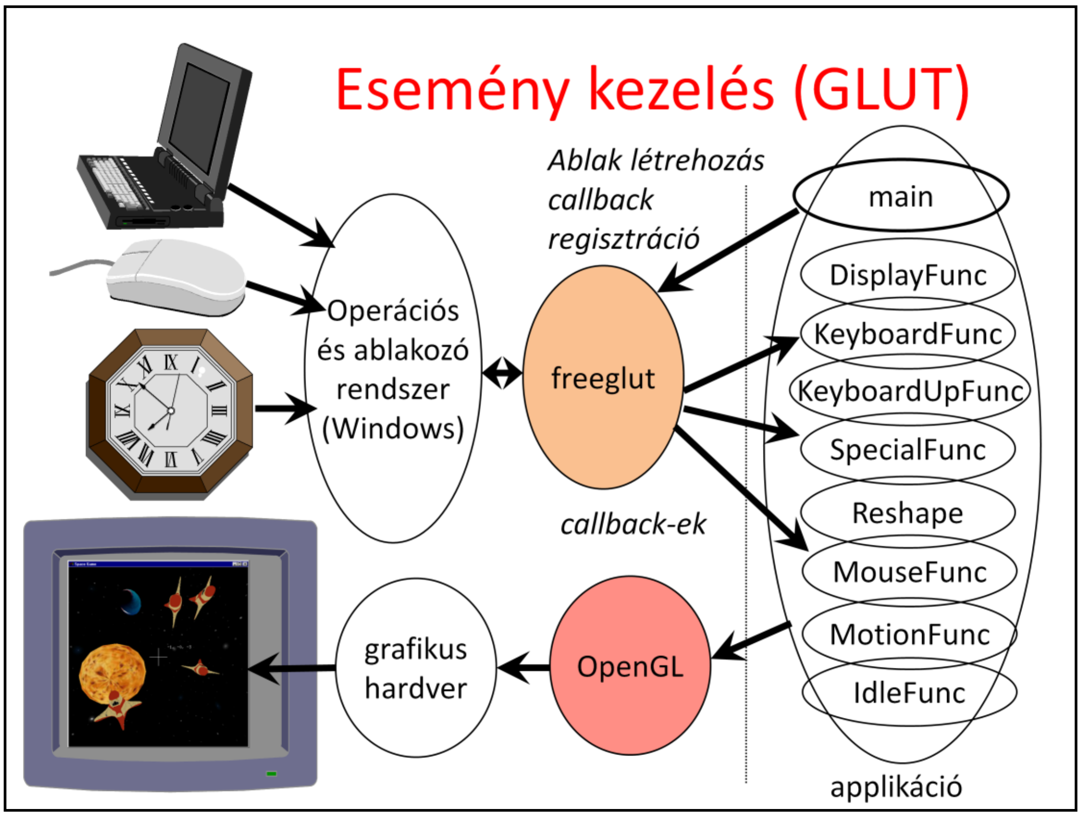
Screen space is the whole screen in full-screen mode or only the application window. The unit is the pixel. Note that screen space is different for the operating system and for OpenGL. For MsWindows and XWindow, axis y points downward while in OpenGL y points upward. Thus, y must be flipped, i.e. subtracted from the vertical resolution. The input pixel coordinate goes from the screen to modeling space, thus inverse transformations are applied in the reverse order. With input devices not only the virtual world can be modified but also the camera can be controlled.



Our graphics application runs under the control of an operating system together with other applications. The operating system handles shared resources like input devices and the frame buffer as well, so a pixel data in the frame buffer can be changed only via the operating system. Modifying pixels one by one from the application would be too slow, so a new hardware element, called the GPU, shows up that is responsible for many time consuming steps of rendering. The GPU is also a shared device that can be accessed via the operating system. Such accesses are calls to a library for the application program. We shall control the GPU through a C graphics library called OpenGL.

On the other hand, to catch input events handled by the operating system, we need another library. We shall utilize the freeGLUT for this purpose, due to its simplicity and the portability (it runs over MsWindows, Xwindow, etc.).

The operating system separates the hardware from the application. The operating system is responsible for application window management and also letting the application give commands to the GPU via OpenGL, not to mention the re-programming of the GPU with shader programs. OpenGL is collection of C functions of names starting with gl. The second part of the name shows what the function does, and the final part allows to initiate the same action with different parameter numbers and types (note that there is not function overloading in C). To get an access to the GPU via OpenGL, the application should negotiate this with the operating system, for which operating system dependent libraries, like the wgl for MsWindows and glX for Xwindow are available. Using them is difficult, and more importantly, it makes our application not portable. So, to hide operating system dependent features, we use GLUT, which translates generic commands to the operating system on which it runs. It is simple and our application will be portable. GLUT function names start with glut.



The graphics output is implemented by OpenGL. The application window management and the input are the responsibilities of GLUT. Our application consists of a main function and a set of event handlers (we use event driven programming paradigm in interactive systems). In main, our application program interacts with GLUT and specifies the properties of the application window (e.g. initial resolution and what a single pixel should store), and also the event handlers.

An event handler is a C function that should be programmed by us. This function is connected to a specific event of GLUT, and having established this connection we expect GLUT to call our function when the specific event occurs. A partial list of possible events are:

- Display event that occurs when the application window becomes invalid and thus GLUT asks the application to redraw the window to restore its content.

- Keyboard event occurs when the user presses a key having ASCII code.

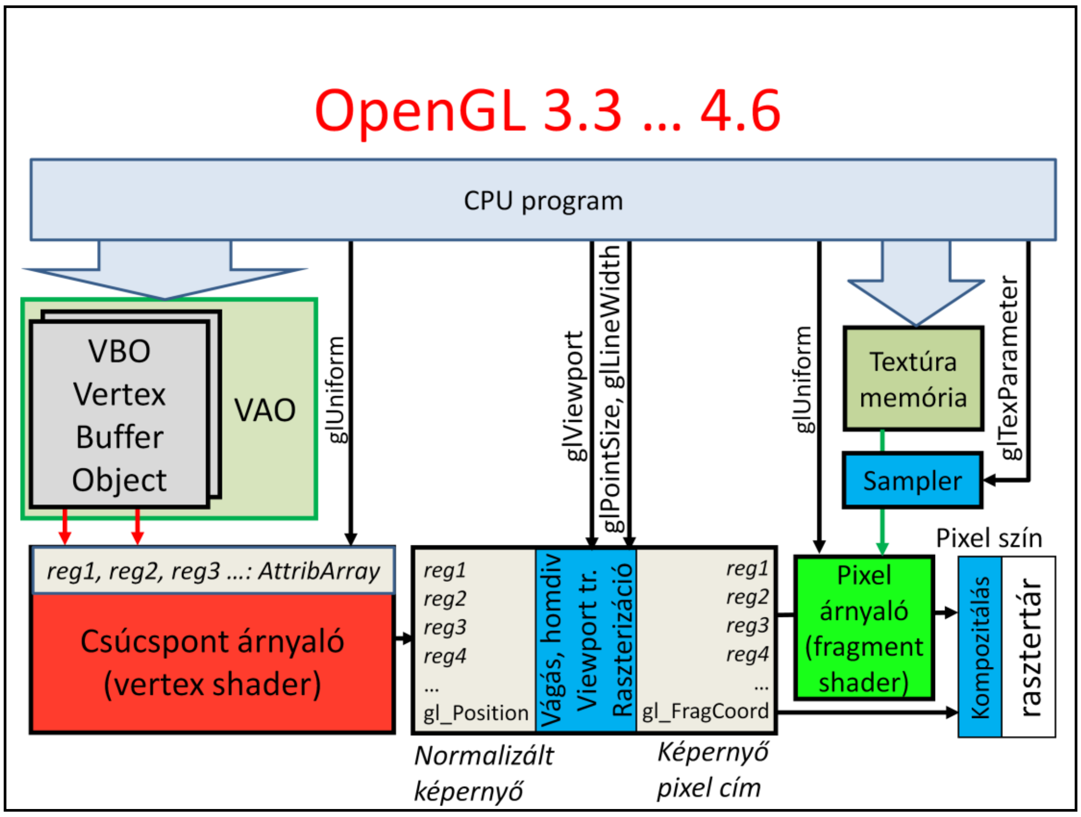
- Special event is like Keyboard event but is triggered by a key press having no ASCII code (e.g. arrows and function keys).

- Reshape handler is called when the dimensions of the application window are changed by the user.

- Mouse event means the pressing or releasing the button of the mouse.

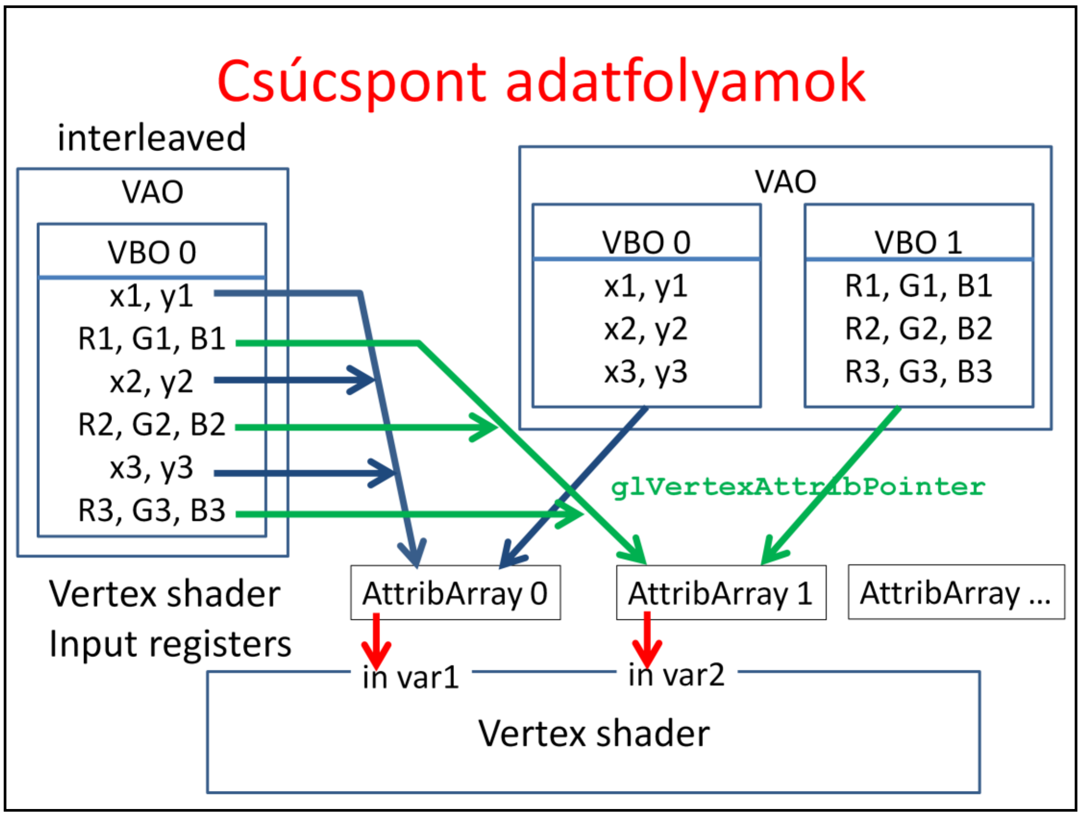
- Idle event indicates the time elapsed and our virtual world model should be updated to reflect the new time.

Event handler registration is optional with the exception of the Display event. If we do not register a handler function, nothing special happens when this event occurs.



Primitives (e.g. a line or a polygon) go down the pipeline, each having multiple vertices associated with their homogeneous coordinates and possible attributes (e.g. vertex color). Primitives must be transformed to normalized device space for clipping, which requires the transformation of its vertices with the modeling, viewing and projection transformation matrices. Clipping is done, so is homogeneous division if the fourth homogeneous coordinate is not 1. Then the primitive is transformed to screen space taking into account the viewport position and size. The primitive is rasterized in screen space.

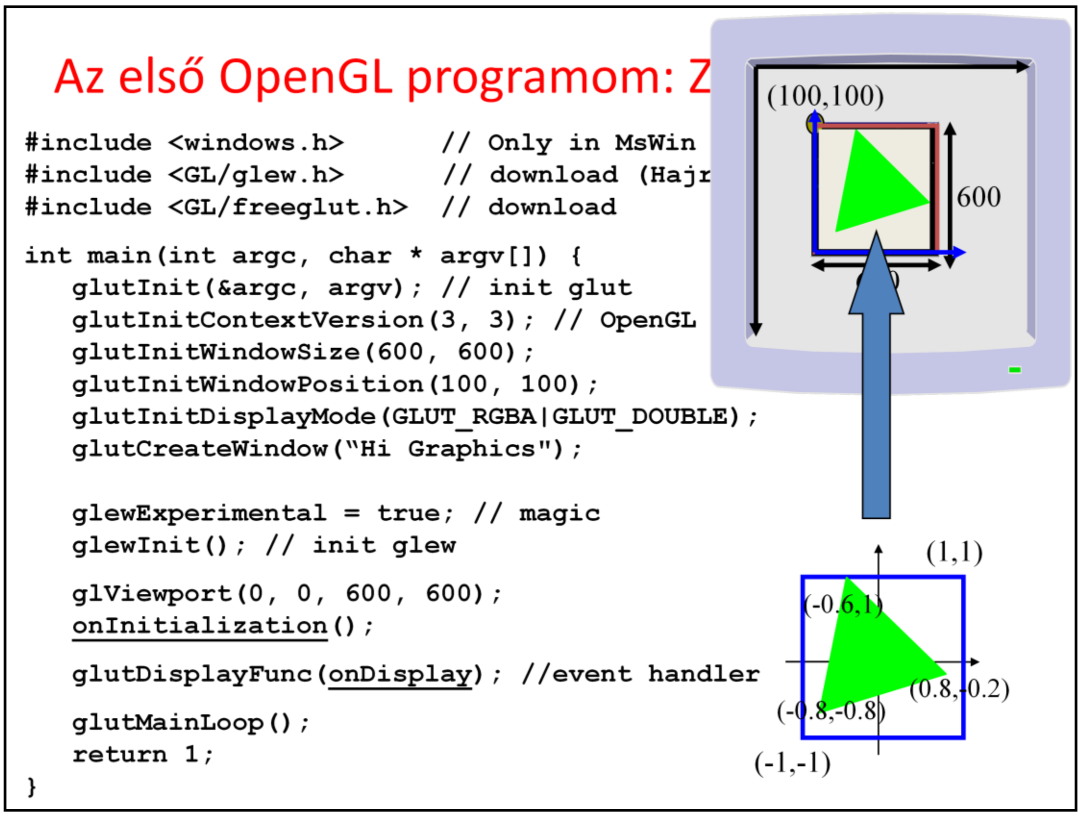
For performance reasons, OpenGL 3 retained mode requires the application to prepare the complete data of the vertices and attributes of a single object rather than passing them one by one. These data are to be stored in arrays on the GPU, called **Vertex Buffer Object** (**VBO**). An object can have multiple VBOs, for example, we can put coordinates in a single array and vertex colors in another. Different VBOs are encapsulated into a **Vertex Array Object** (**VAO**) that also stores information about how the data should be fetched from the VBOs and sent to the input registers of the **Vertex Shader**. A single input register can store four 32 bit long words (4 floats called vec4, or four integers) and is called **Vertex Attrib Array.** The responsibility of the Vertex Shader is to transform the object to normalized device space. If the concatenation of model, view and projection matrices is given to the Vertex Shader, it is just a single matrix-vector multiplication. The output of the Vertex Shader goes to output registers including gl_Position storing the vertex position in normalized device space and other registers storing vertex attributes. Clipping, homogeneous division, viewport transformation and rasterization are performed by the **fixed function hardware** of the GPU, so these steps cannot be programmed. The output of the rasterization step is the sequence of pixels with pixel coordinates and interpolated vertex attributes. Pixel coordinates select the pixel that is modified in the frame buffer. From other vertex attributes and global variables, the pixel color should be computed by another programmable unit called the **Fragment Shad**er.



Let us zoom out the connection of the vertex buffer objects, vertex shader input registers called AttribArrays, and vertex shader input variables. The object is described in arrays called VBOs. For example, coordinates can be stored in one array, colors in another (this strategy is called Structure Of Arrays, or SOA for short). To allow the Vertex Shader to process a vertex, its input registers must be filled with the data of that particular vertex, one vertex at a time. Function **glVertexAttribPointer** tells the GPU how to interpret the data in VBOs, from where the data of a single vertex can be fetched, and in which AttribArray a data element should be copied. For example, coordinates can be copied to AttribArray0 while colors to AttribArray1 (a single register can store 4 floats).

When the Vertex Shader runs, it can fetch its input registers. It would not be too elegant if we had to refer to the name of the input register, e.g. AttribArray 0, so it is possible to assign variable names to it. For example, AttribArray0 can be the ”vertexPosition”.

Note that this was only one possibility of data organization. For example, it is also perfectly reasonable to put all data in a single array where coordinates and attributes of a single vertex are not separated (this strategy is the Array Of Structures, or AOS). In this case glVertexAttribPointer should tell the GPU where an attribute starts in the array and what the step size (stride) is.



Az első OpenGL programom: Z

```
#include <windows.h>      // Only in MsWin
#include <GL/glew.h>      // download (Hajr
#include <GL/freeglut.h>  // download

int main(int argc, char * argv[]) {
   glutInit(&argc, argv); // init glut
   glutInitContextVersion(3, 3); // OpenGL
   glutInitWindowSize(600, 600);
   glutInitWindowPosition(100, 100);
   glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);
   glutCreateWindow("Hi Graphics");

   glewExperimental = true; // magic
   glewInit(); // init glew

   glViewport(0, 0, 600, 600);
   onInitialization();

   glutDisplayFunc(onDisplay); //event handler

   glutMainLoop();
   return 1;
}
```

In the main function of a graphics application, we set up the application window with the help of GLUT telling the initial position, size, what data should be stored in a pixel, and also what functions should be called when different events happen. At the end, the message loop is started, which runs in circles, checks whether any event occurred for which we have registered an event handler, and if this is the case, it calls the respective event handler.

The main function can also be used to initialize data in OpenGL (on the GPU), especially those which are needed from the beginning of the program execution and which do not change during the application. We need shader programs from the beginning, so this is a typical place to compile and link shader programs and upload them to the GPU.

Let us start with the main function. Two header files are needed. GLEW is the OpenGL Extension Wrangler library that finds out what extensions are supported by the current GPU in run time. GLUT is a windowing utility toolkit to set up the application window and to manage events. In the main functions, first the application window is set up with glut calls:

- glutInit initializes glut and allows use to communicate with the GPU via OpenGL.

- glutInitContextVersion sets the required OpenGL version. In this case, we want opengl 3.0.

- glutInitWindowSize specifies the initial resolution of the application window.

- glutInitWindowPosition specifies where it is initially placed relative to the upper left corner of the screen.

- glutInitDisplayMode tells glut what to store in a single pixel. In the current case, we store 8 bit (default) R,G,B, and A (opacity) values, in two copies to support double buffering.

- glutCreateWindow creates the window, which shows up.

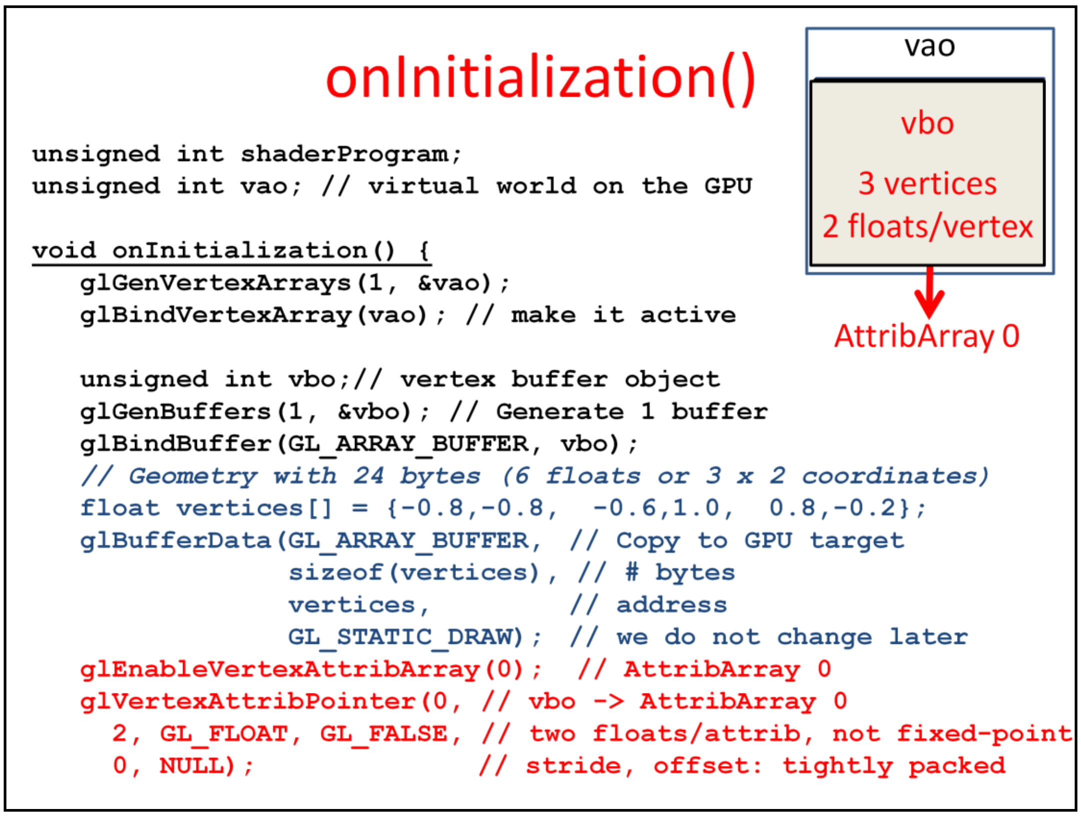
The Extension Wrangler is initialized

- glewExperimental = true: GLEW obtains information on the supported extensions from the graphics driver, so if it is not updated, then it might not report all features the GPU can deliver. Setting glewExperimental to true gets GLEW to try the extension even if it is not listed by the driver.
- glewInit makes the initialization

From here, we can initialize OpenGL.

- glViewport sets the render target, i.e. the photograph inside the application window
- onInitialization is our custom initialization function discussed on the next slide.

The remaining functions register event handlers and start the message loop. For the time being, only the onDisplay is relevant, which is called whenever the application window becomes invalid. We use this function to render the virtual world, which consists of a single green triangle, directly given in normalized device space.



onInitialization()

```
unsigned int shaderProgram;
unsigned int vao; // virtual world on the GPU

void onInitialization() {
   glGenVertexArrays(1, &vao);
   glBindVertexArray(vao); // make it active

   unsigned int vbo;// vertex buffer object
   glGenBuffers(1, &vbo); // Generate 1 buffer
   glBindBuffer(GL_ARRAY_BUFFER, vbo);
   // Geometry with 24 bytes (6 floats or 3 x 2 coordinates)
   float vertices[] = {-0.8,-0.8,  -0.6,1.0,  0.8,-0.2};
   glBufferData(GL_ARRAY_BUFFER,  // Copy to GPU target
                sizeof(vertices), // # bytes
                vertices,         // address
                GL_STATIC_DRAW);  // we do not change later
   glEnableVertexAttribArray(0);  // AttribArray 0
   glVertexAttribPointer(0, // vbo -> AttribArray 0
     2, GL_FLOAT, GL_FALSE, // two floats/attrib, not fixed-point
     0, NULL);              // stride, offset: tightly packed
```

In onInitialization those opengl data are initialized that are typically constant during the application, so they do not have to be set in every drawing. In our program, this includes the constant geometry (the triangle), and the GPU shader programs. The shaderProgram and the vao are set here but also used in the onDisplay, therefore they are global variables.

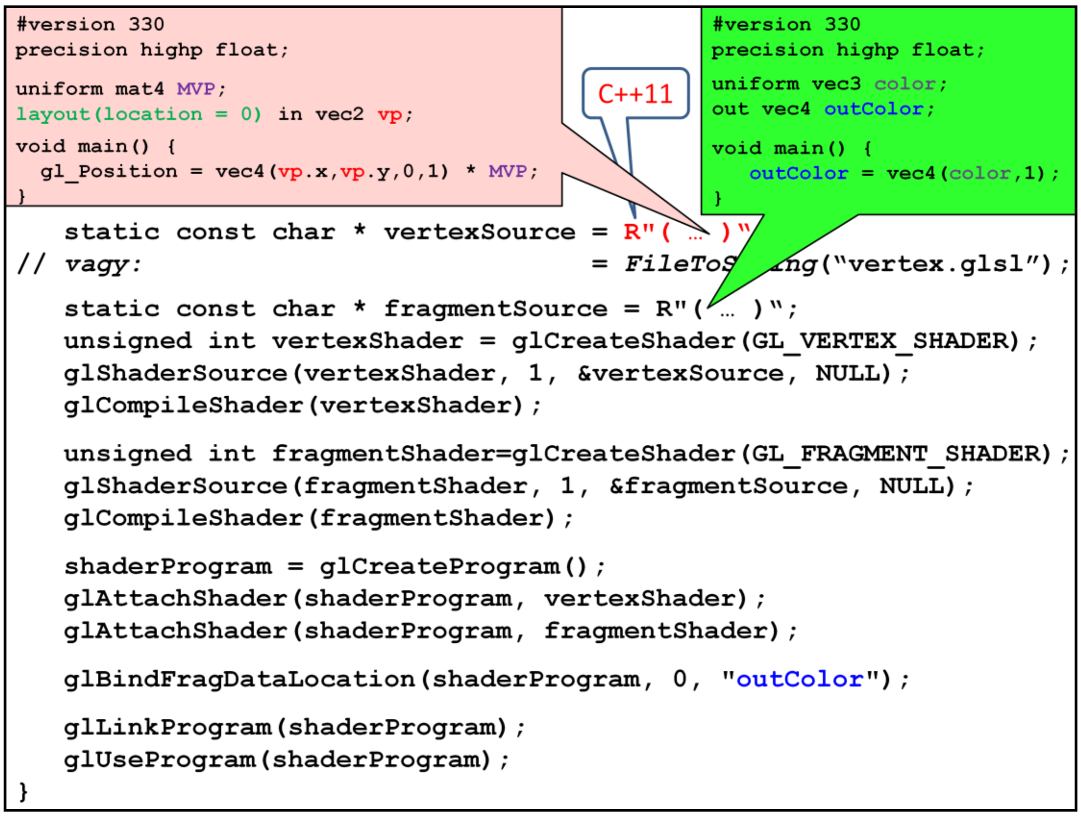
First we allocate one vertex array object and its id is vao. With Binding, this is made active, which means that all subsequent operations belong to this vao until another vao is bound or the current one is unbound with binding 0.

In the second step, one vertex buffer object is allocated, which will be part of the active vao (we have just one, which is active). This vertex buffer object is made active, so all subsequent operations are related to this until another is bound.

Array ”vertices” stores the geometry of our triangle, and is obviously in the CPU memory. It contains 6 floats, i.e. 24 bytes. With **glBufferData** the 24 bytes are copied to the GPU. With the last parameter of glBufferData we can specify which type of GPU memory should be used (the GPU has different types of memory with different write and read speeds and capacity, so the driver may decide where to copy this 24 bytes based on our preference). We say that the 24 bytes will not be modified but it would be great if it could be fetched fast (constant memory would be an ideal choice). So far we said nothing about the organization and the meaning of the data, it is simply 24 bytes, the GPU does not know that it defines 3 vertices, each with 2 Cartesian coordinates, which are in float format.

glVertexAttribPointer() defines the interpretation of the data and also that the data associated with a single vertex goes to the input register (AttribArray) number 0. It specifies that a single vertex have two floats, i.e. 8 bytes. If it was non floating point value, it would also be possible to put the binary point to the most significant bit, but we set this parameter to GL_FALSE.

The last two parameters tell the GPU how many bytes should be stepped from one vertex to the other (if it is 0, it means that the step size is equal to the data size, 2 floats in this case), and where the first element is (at the beginning of the array, so the pointer offset is zero). Stride and offset are essential if interleaved vbos are used.


```
#version 330
precision highp float;
uniform mat4 MVP;
layout(location = 0) in vec2 vp;
void main() {
  gl_Position = vec4(vp.x,vp.y,0,1) * MVP;
}
```

```
#version 330
precision highp float;
uniform vec3 color;
out vec4 outColor;
void main() {
   outColor = vec4(color,1);
}
```

```
   static const char * vertexSource = R"( … )";
// vagy:                             = FileToS   ng("vertex.glsl");
   static const char * fragmentSource = R"( … )";
   unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
   glShaderSource(vertexShader, 1, &vertexSource, NULL);
   glCompileShader(vertexShader);

   unsigned int fragmentShader=glCreateShader(GL_FRAGMENT_SHADER);
   glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
   glCompileShader(fragmentShader);

   shaderProgram = glCreateProgram();
   glAttachShader(shaderProgram, vertexShader);
   glAttachShader(shaderProgram, fragmentShader);

   glBindFragDataLocation(shaderProgram, 0, "outColor");

   glLinkProgram(shaderProgram);
   glUseProgram(shaderProgram);
}
```

The remaining part of the onInitialization gets the shader programs ready. The source of the shader programs can be read from a file or directly copied from a string. We use here the latter option. As programs are typically written in more than one line, the string cannot be simple "…" but should be special and hold new line characters, which is possible with the R"(…)" C++ feature.

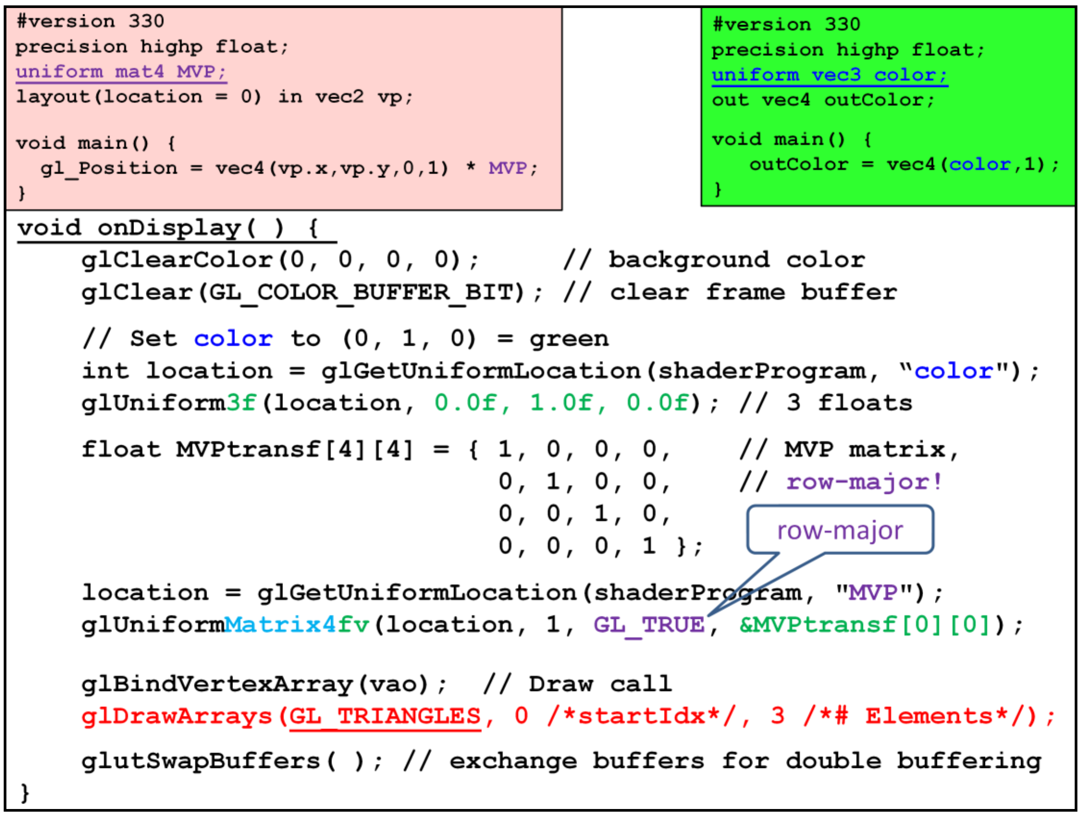
The vertex shader source code starts with the version number that tells the compiler how matured GPU is assumed during execution.

Uniform parameters are like constants that cannot change during the drawing of a single primitive. MVP is a 4x4 matrix (type mat4), which represents the model-view-projection matrix. The vertex shader has one per-vertex attribute, defined with variable name vp and storing the x, y coordinates of the current vertex. The vertex shader code computes the multiplication of 4 element vector that is the conversion of vp to 3D homogeneous coordinates and the 4x4 MVP matrix, and the result is written into a specific output register called gl_Position, which should get the point transformed to normalized device space. The vertex shader could output other variables as well, which would follow the point during clipping and rasterization, and would be interpolated during these operations. Clipping, homogeneous division, viewport transform and rasterization are fixed function elements that cannot be programmed.

The output of the fixed function part is the sequence of pixels (called fragments) belonging to the current primitive and also the variables that are output by the vertex shader, having interpolated for the current pixel. The pixel address is in register gl_FragCoord, which cannot be modified, but from the other registers and uniform variables, the color of this fragment can be obtained by the fragment shader processor. It has one uniform input called the color, which will determine the output color stored in variable outColor.

The very beginning of the pipeline, vertex coordinate variable vp is connected to the vertex shader input register (AttribArray) number 0 as told by **glBindAttribLocation**. The output of the fragment shader goes to the frame buffer as requested by **glBindFragDataLocation**.

Finally, the shader program is linked, copied to the shader processors to be executed by them.



```
#version 330
precision highp float;
uniform mat4 MVP;
layout(location = 0) in vec2 vp;

void main() {
  gl_Position = vec4(vp.x,vp.y,0,1) * MVP;
}
```

```
#version 330
precision highp float;
uniform vec3 color;
out vec4 outColor;

void main() {
   outColor = vec4(color,1);
}
```

```
void onDisplay( ) {
    glClearColor(0, 0, 0, 0);     // background color
    glClear(GL_COLOR_BUFFER_BIT); // clear frame buffer

    // Set color to (0, 1, 0) = green
    int location = glGetUniformLocation(shaderProgram, "color");
    glUniform3f(location, 0.0f, 1.0f, 0.0f); // 3 floats

    float MVPtransf[4][4] = { 1, 0, 0, 0,    // MVP matrix,
                              0, 1, 0, 0,    // row-major!
                              0, 0, 1, 0,
                              0, 0, 0, 1 };

    location = glGetUniformLocation(shaderProgram, "MVP");
    glUniformMatrix4fv(location, 1, GL_TRUE, &MVPtransf[0][0]);

    glBindVertexArray(vao);  // Draw call
    glDrawArrays(GL_TRIANGLES, 0 /*startIdx*/, 3 /*# Elements*/);

    glutSwapBuffers( ); // exchange buffers for double buffering
}
```

We have registered a single event handler (onDisplay) that reacts to the event occurring when the application window gets invalid (DisplayFunc).

In this function, the virtual world (consisting of the single green triangle) is rendered.

glClearColor sets the color with which the pixels of the application window is cleared. This color is black. The actual clearing is done by glClear. **GL_COLOR_BUFFER_BIT** stands for the frame buffer storing color values.

Drawing consists of setting the values of uniform variables of shaders and then forcing the geometry through the pipeline, which is called the draw call. Finally, the buffer used for drawing so far is swapped with the buffer the user could see so far by **glutSwapBuffers**, so the result will be visible to the user.

The fragment shader has a single uniform variable called color and of type vec3, which can be set with function **glUniform3f(location, 0.0f, 1.0f, 0.0f)** to value (0,1,0)=green. Note that ”3f” at the end of the function name indicates that this function takes 3 float parameters. Parameter location is the serial number of this uniform variable, which can be found with **glGetUniformLocation(shaderProgram, “color")** which returns the serial number of uniform variable called “color” in the shader programs.

The vertex shader has uniform variable MVP of type mat4, i.e. it is a 4x4 matrix. First, its serial number must be obtained, then its value can be set with **glUniformMatrix4fv**. Here fv means that matrix elements are floats (f) and instead of passing the 16 floats by value, the address of the CPU array is given (v) from which the values can be copied (pass by address). The second parameter of **glUniformMatrix4fv** says that 1 matrix is passed, the third parameter that this is a row-major matrix and should be kept this way.

This issue can cause a lot of confusion:

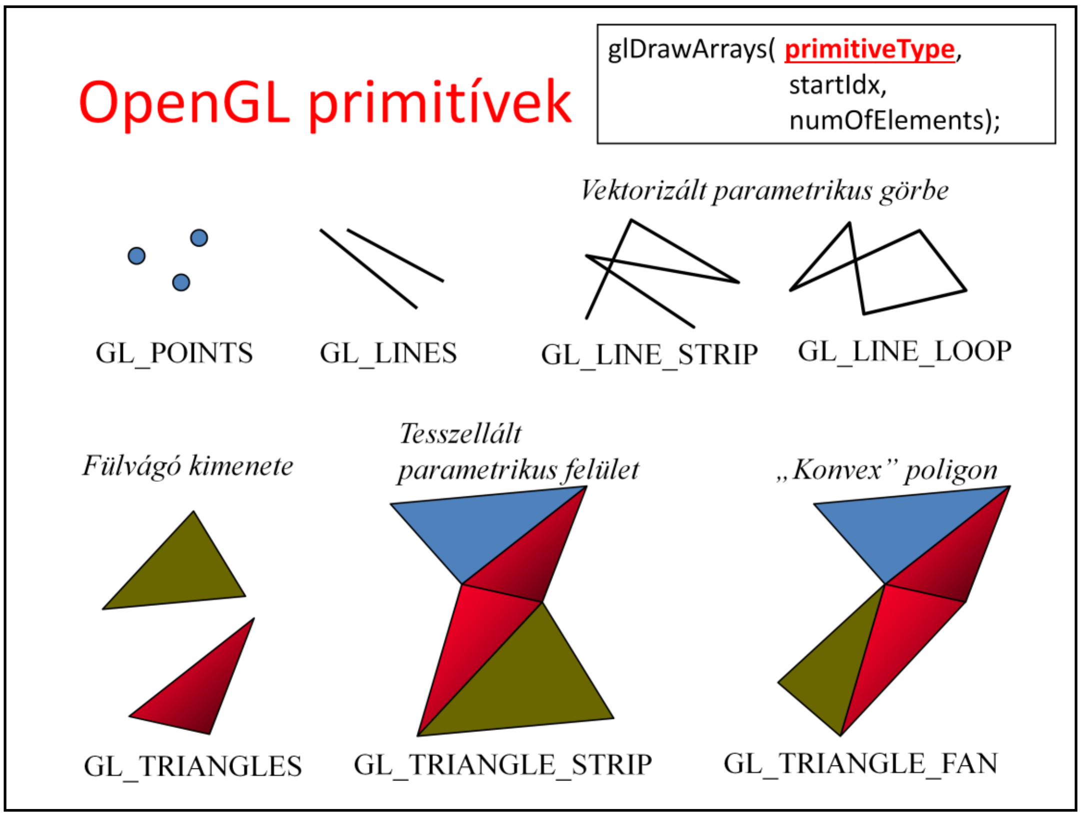
- In C or C++ two-dimensional matrices are of row-major.
- In GLSL two-dimensional matrices are of column-major.

So if we use them without caution, we might apply the transpose of the matrix and not what we wanted. There are many solutions for this problem:

1. Use an own 2D matrix class in C++ that follows the column-major indexing scheme, and consider vectors of points as row vectors both on the CPU and on the GPU.
2. Use an own 2D matrix class in C++ that follows the column-major indexing scheme, and consider vectors of points as column vectors both on the CPU and on the GPU. The matrices will be transposed with respect to the previous solution.
3. Use the standard 2D matrix indexing on the CPU (row-major) and the standard 2D matrix indexing on the GPU (column-major), but consider points as row vector in the CPU program (and therefore put on the left side of the transformation matrix) and column vector in the GPU program (and put on the right side of the matrix).
4. Use the standard 2D matrix indexing on the CPU (row-major) but transpose the matrix when passed to the GPU, and consider points as row vector both in the CPU program and in the GPU program.

We use option 4, and setting the third parameter of **glUniformMatrix4fv** to TRUE enables just the required transpose.

Vertex Array Objects are our virtual world objects already uploaded to the GPU. With **glBindVertexArray(vao)** we can select one object for subsequent operations (drawing) and finally **glDrawArrays** gets the current VAO to feed the pipeline, i.e. this object is rendered. We may not send all vertices of this object, so with startIdx and number of elements a subset can be selected. Setting startIdx to 0 and sending all 3 points, our whole triangle is rendered. The first parameter of **glDrawArrays** tells the GPU the topology of the primitive, that is, what the vertices define. In our case, triangles, and as we have only 3 vertices, a single triangle.

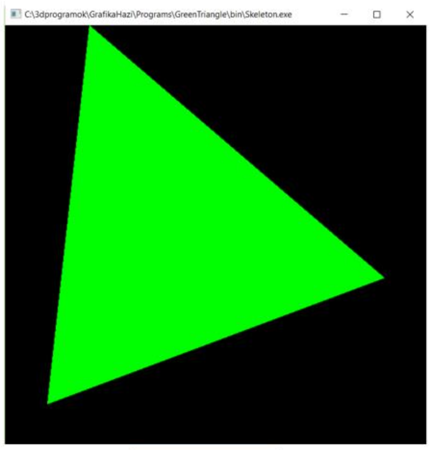


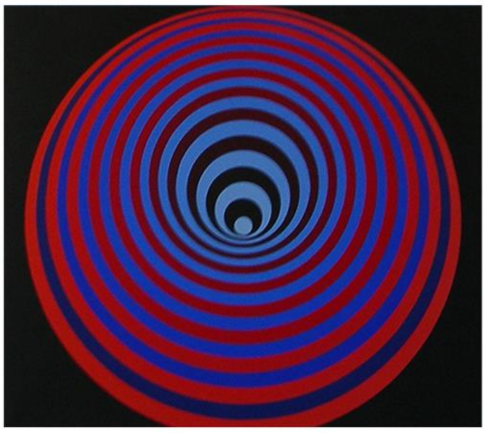
OpenGL primitívek

```
glDrawArrays( primitiveType,
              startIdx,
              numOfElements);
```

This is the set of possible primitive types. Basically points, line segments and triangles, but in sophisticated options sharing vertices is also possible.

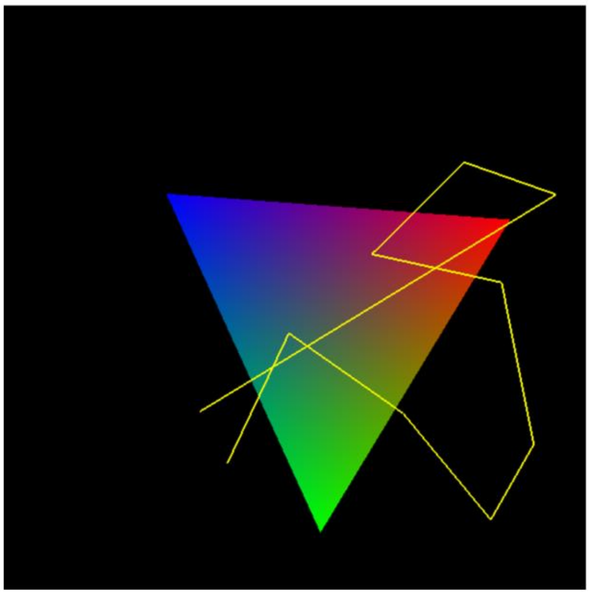
Zöld háromszög versus Victor Vasarely


C:\3dprogramok\GrafikaHazi\Programs\GreenTriangle\bin\Skeleton.exe

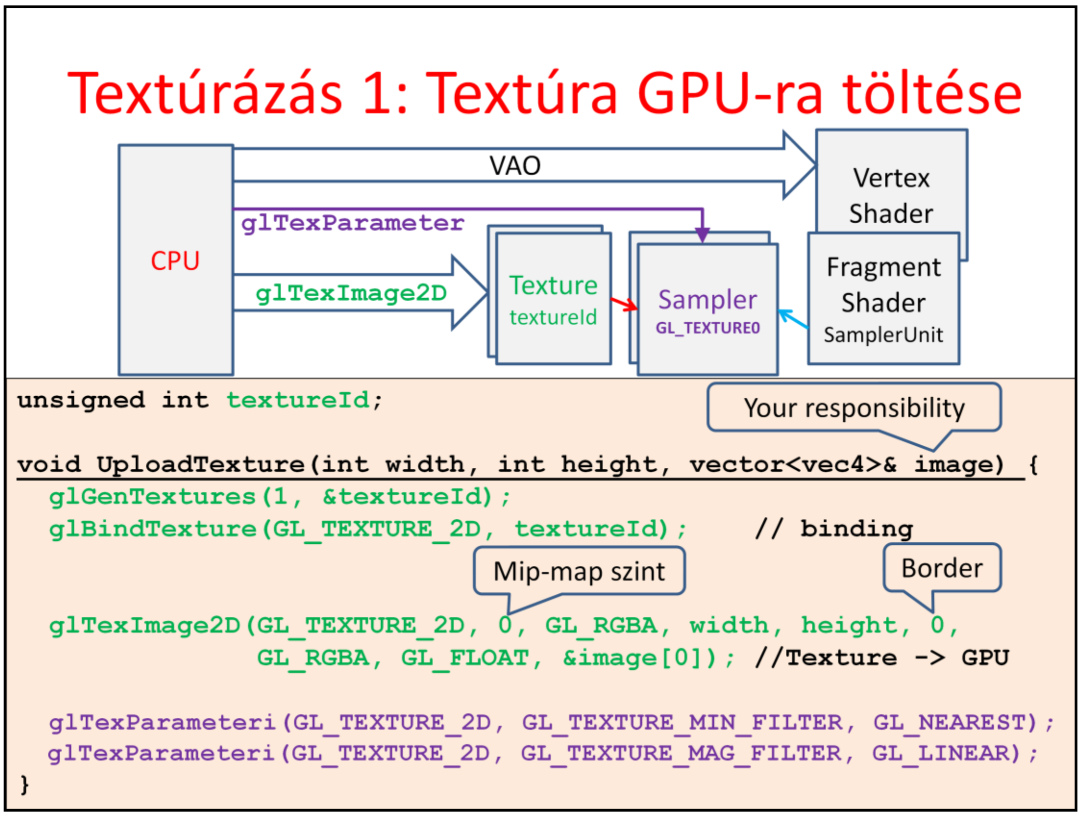


Having a program that draws a green triangle, it is easy to produce Vasarely like graphics.

Csúcspont attribútumok, dinamikus világ, világ-kép szétválasztás, interakció



In this demo the triangle is rotating, the camera can be panned and zoomed, and we can interactively define a polyline.

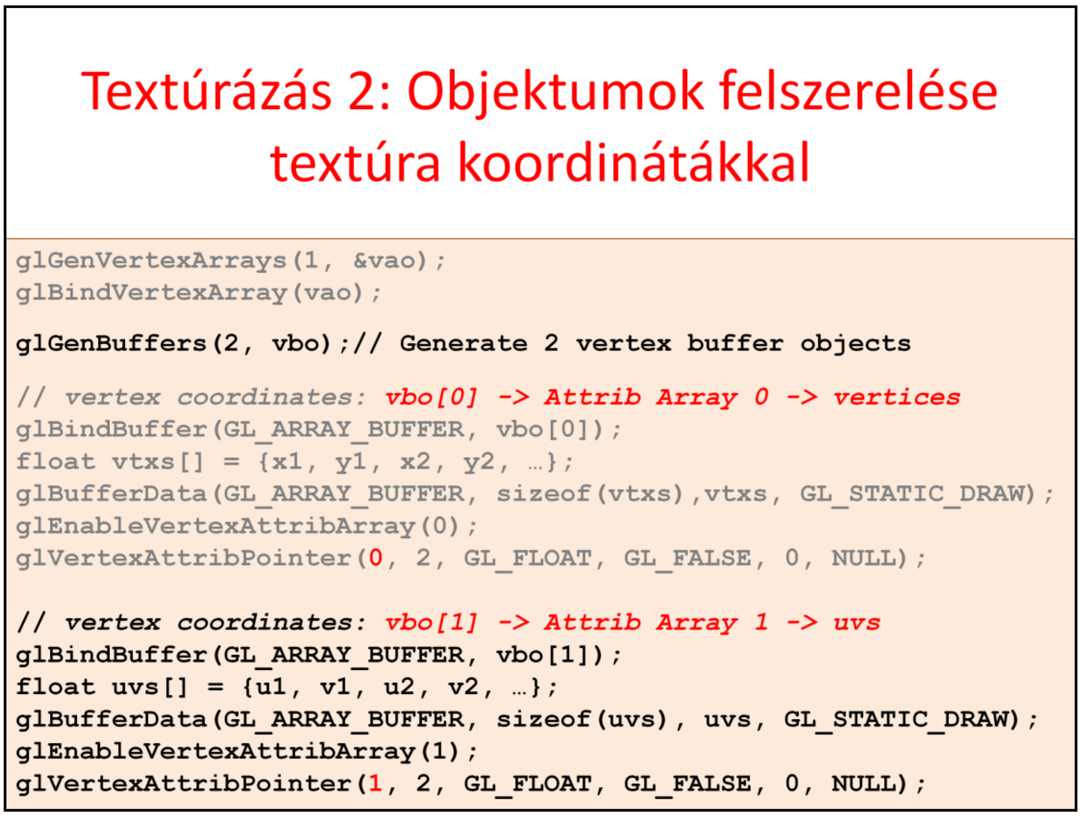


Textúrázás 1: Textúra GPU-ra töltése

```
unsigned int textureId;

void UploadTexture(int width, int height, vector<vec4>& image) {
   glGenTextures(1, &textureId);
   glBindTexture(GL_TEXTURE_2D, textureId);    // binding

   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
                GL_RGBA, GL_FLOAT, &image[0]); //Texture -> GPU

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

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.

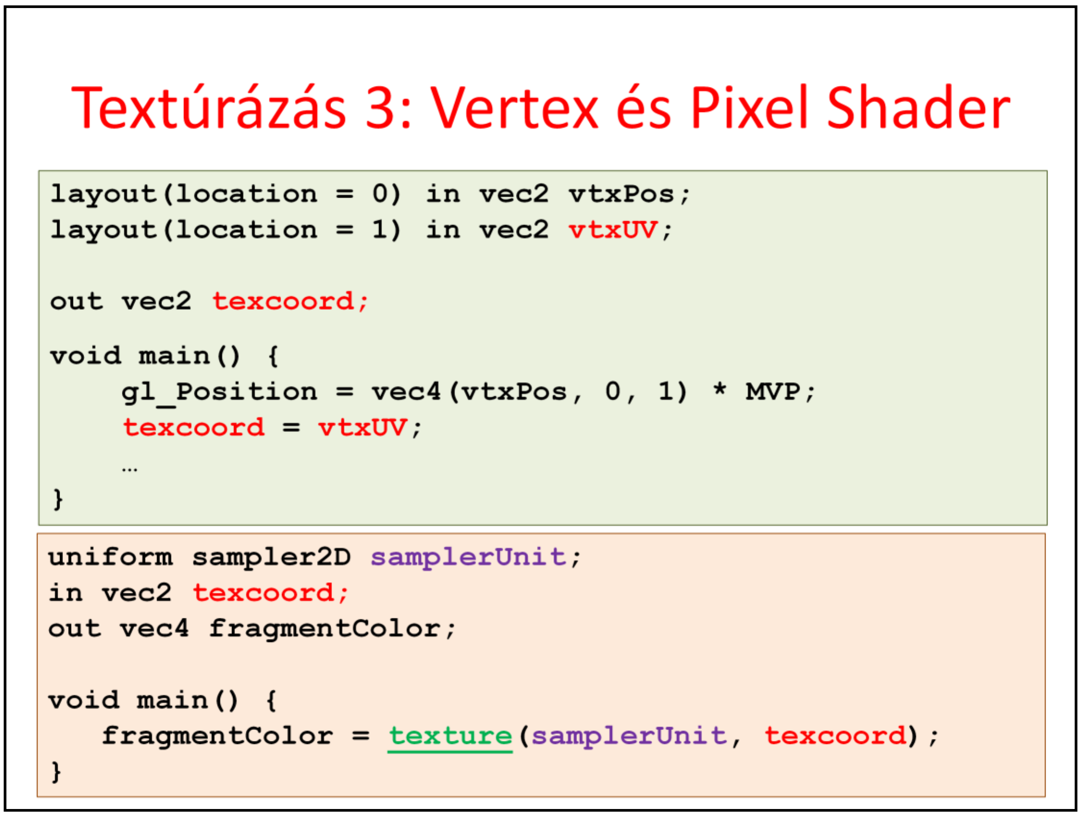


Textúrázás 2: Objektumok felszerelése textúra koordinátákkal

```
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.



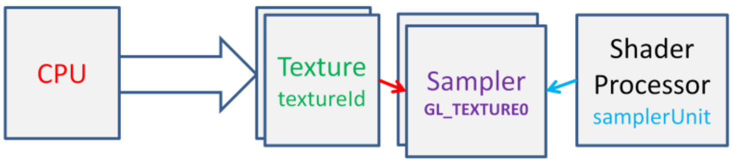
Textúrázás 3: Vertex és Pixel Shader

```
layout(location = 0) in vec2 vtxPos;
layout(location = 1) in vec2 vtxUV;

out vec2 texcoord;

void main() {
    gl_Position = vec4(vtxPos, 0, 1) * MVP;
    texcoord = vtxUV;
    …
}
```

```
uniform sampler2D samplerUnit;
in vec2 texcoord;
out vec4 fragmentColor;

void main() {
   fragmentColor = texture(samplerUnit, texcoord);
}
```

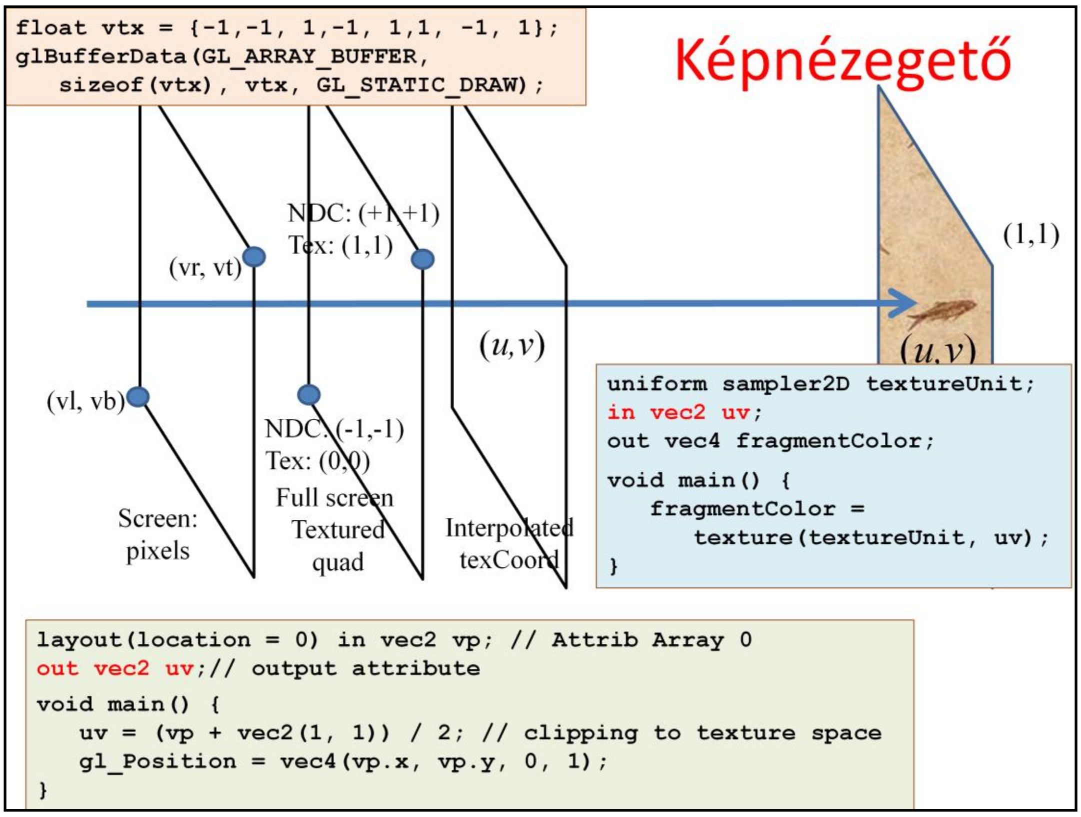
The vertex shader gets the location of the point and also the uv associated with this point. The location is transformed to normalized device space, the texture coordinate is only copied by the vertex shader. The fixed function hardware interpolates the texture coordinates and the fragment shader looks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



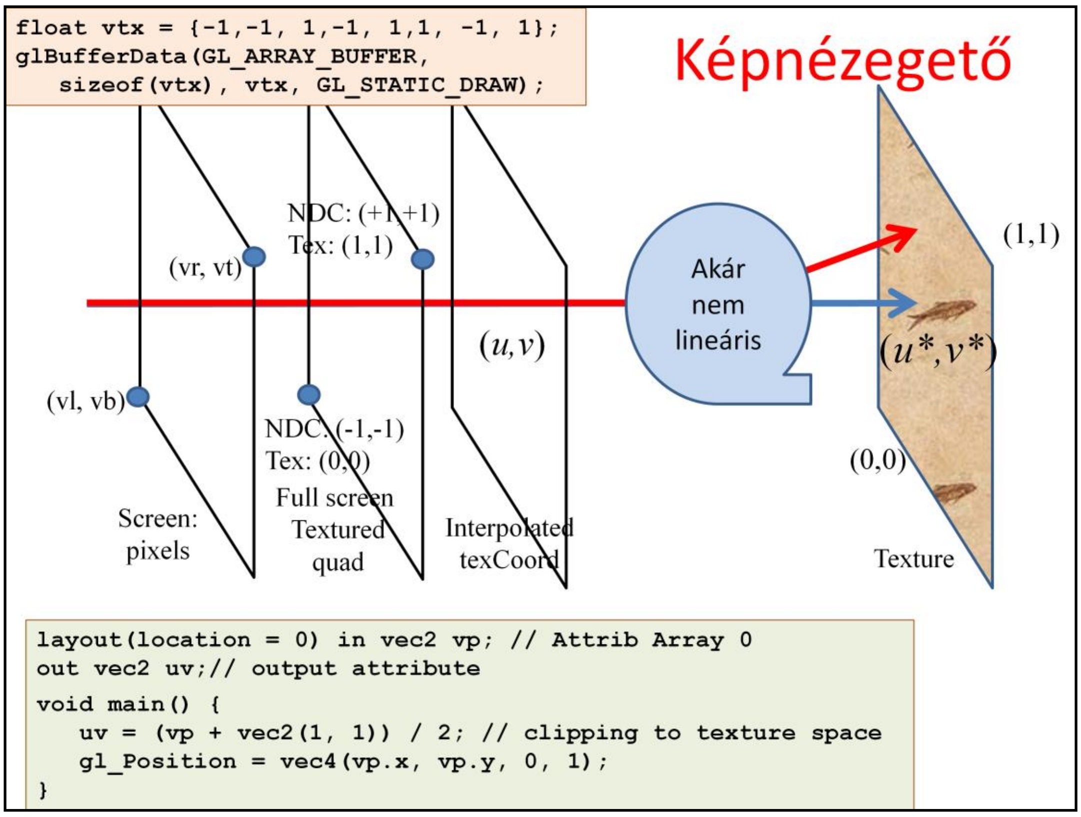
Képnézegető

```
float vtx = {-1,-1, 1,-1, 1,1, -1, 1};
glBufferData(GL_ARRAY_BUFFER,
    sizeof(vtx), vtx, GL_STATIC_DRAW);
```

```
uniform sampler2D textureUnit;
in vec2 uv;
out vec4 fragmentColor;

void main() {
   fragmentColor =
      texture(textureUnit, uv);
}
```

```
layout(location = 0) in vec2 vp; // Attrib Array 0
out vec2 uv;// output attribute

void main() {
   uv = (vp + vec2(1, 1)) / 2; // clipping to texture space
   gl_Position = vec4(vp.x, vp.y, 0, 1);
}
```

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.



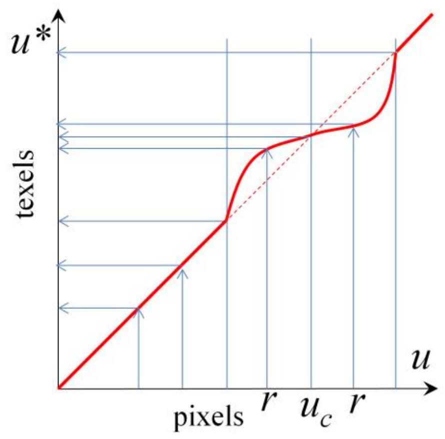
Képnézegető

```
float vtx = {-1,-1, 1,-1, 1,1, -1, 1};
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vtx), vtx, GL_STATIC_DRAW);
```

```
layout(location = 0) in vec2 vp; // Attrib Array 0
out vec2 uv;// output attribute
void main() {
   uv = (vp + vec2(1, 1)) / 2; // clipping to texture space
   gl_Position = vec4(vp.x, vp.y, 0, 1);
}
```

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

Mágikus lencse



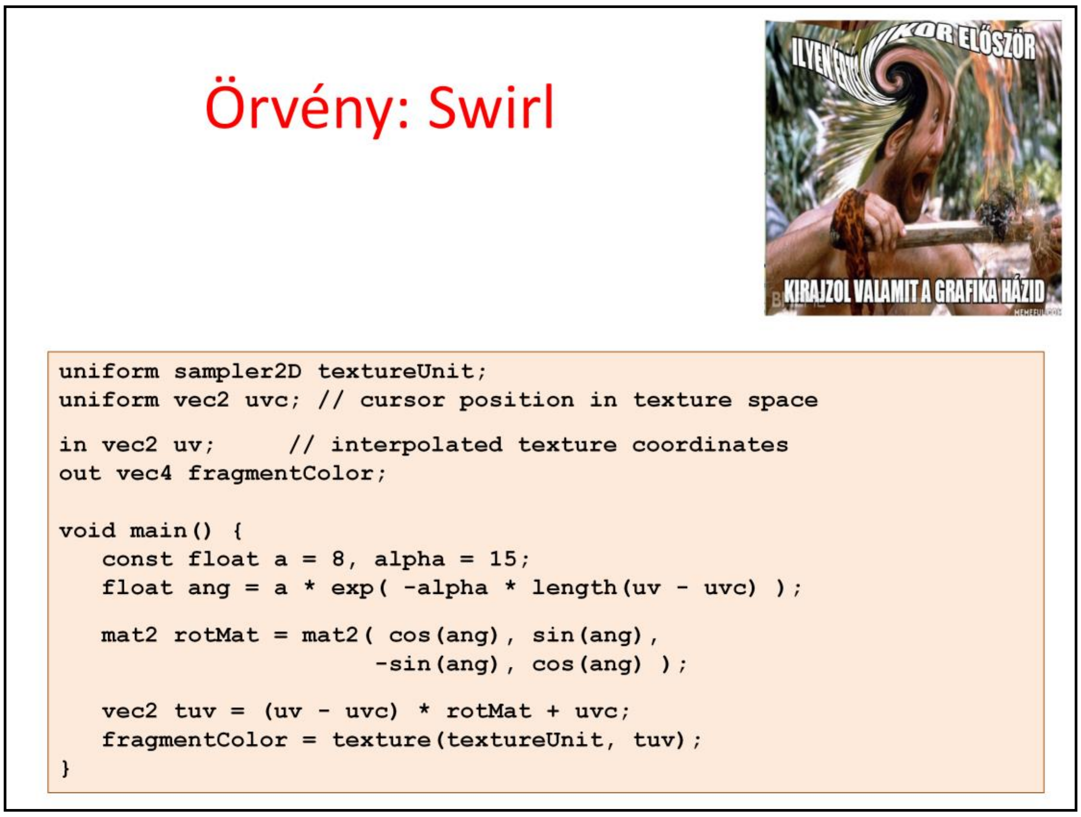
$$\boxed{u^* = \frac{(u-u_c)^3}{r^2} + u_c \quad \text{if } |u-u_c| < r}$$

```
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.

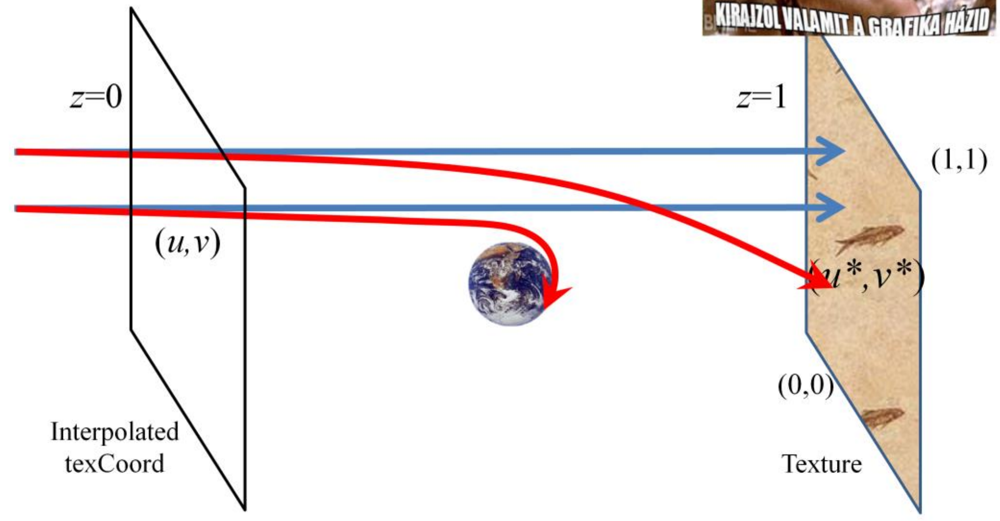


Örvény: Swirl

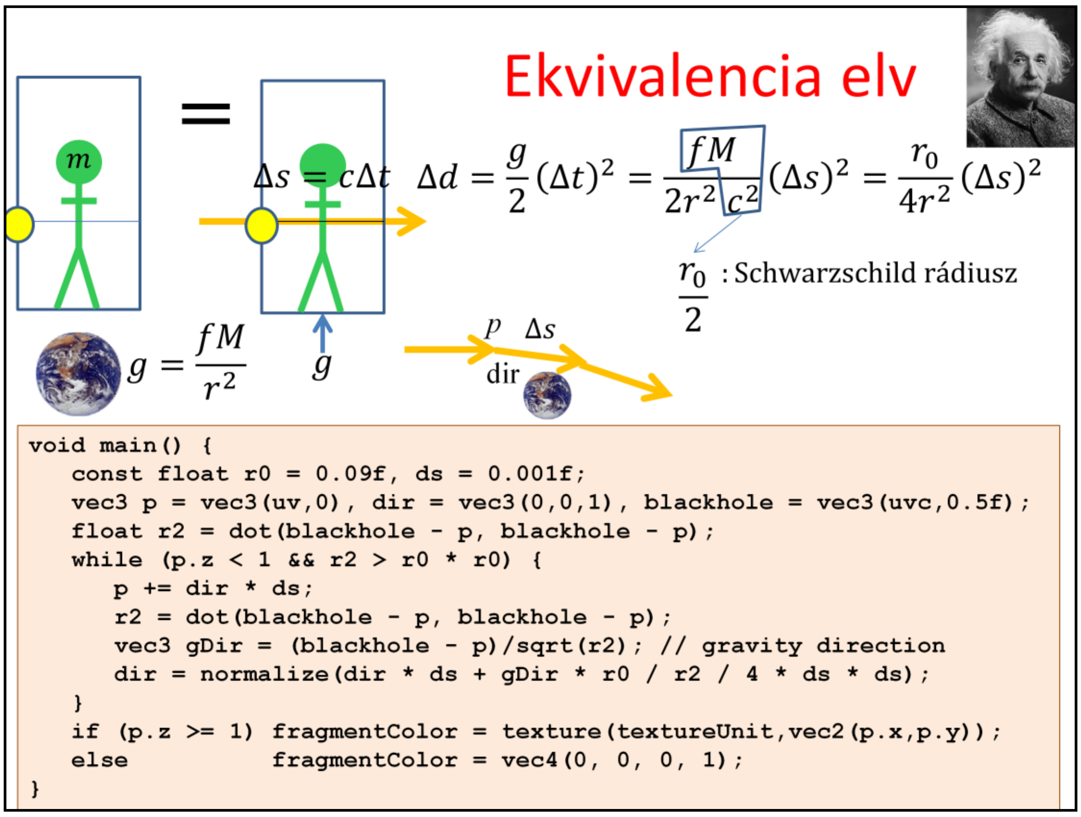
```
uniform sampler2D textureUnit;
uniform vec2 uvc; // cursor position in texture space

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.

Gravitáció (fekete lyukak)



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.



Ekvivalencia elv

$$\Delta s = c\Delta t \quad \Delta d = \frac{g}{2}(\Delta t)^2 = \frac{\boxed{fM}}{2r^2 \boxed{c^2}}(\Delta s)^2 = \frac{r_0}{4r^2}(\Delta s)^2$$

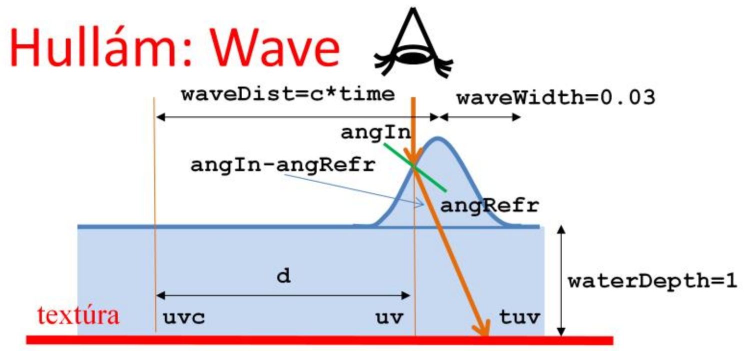
$\frac{r_0}{2}$: Schwarzschild rádiusz

$$g = \frac{fM}{r^2}$$

```
void main() {
   const float r0 = 0.09f, ds = 0.001f;
   vec3 p = vec3(uv,0), dir = vec3(0,0,1), blackhole = vec3(uvc,0.5f);
   float r2 = dot(blackhole - p, blackhole - p);
   while (p.z < 1 && r2 > r0 * r0) {
      p += dir * ds;
      r2 = dot(blackhole - p, blackhole - p);
      vec3 gDir = (blackhole - p)/sqrt(r2); // gravity direction
      dir = normalize(dir * ds + gDir * r0 / r2 / 4 * ds * ds);
   }
   if (p.z >= 1) fragmentColor = texture(textureUnit,vec2(p.x,p.y));
   else          fragmentColor = vec4(0, 0, 0, 1);
}
```

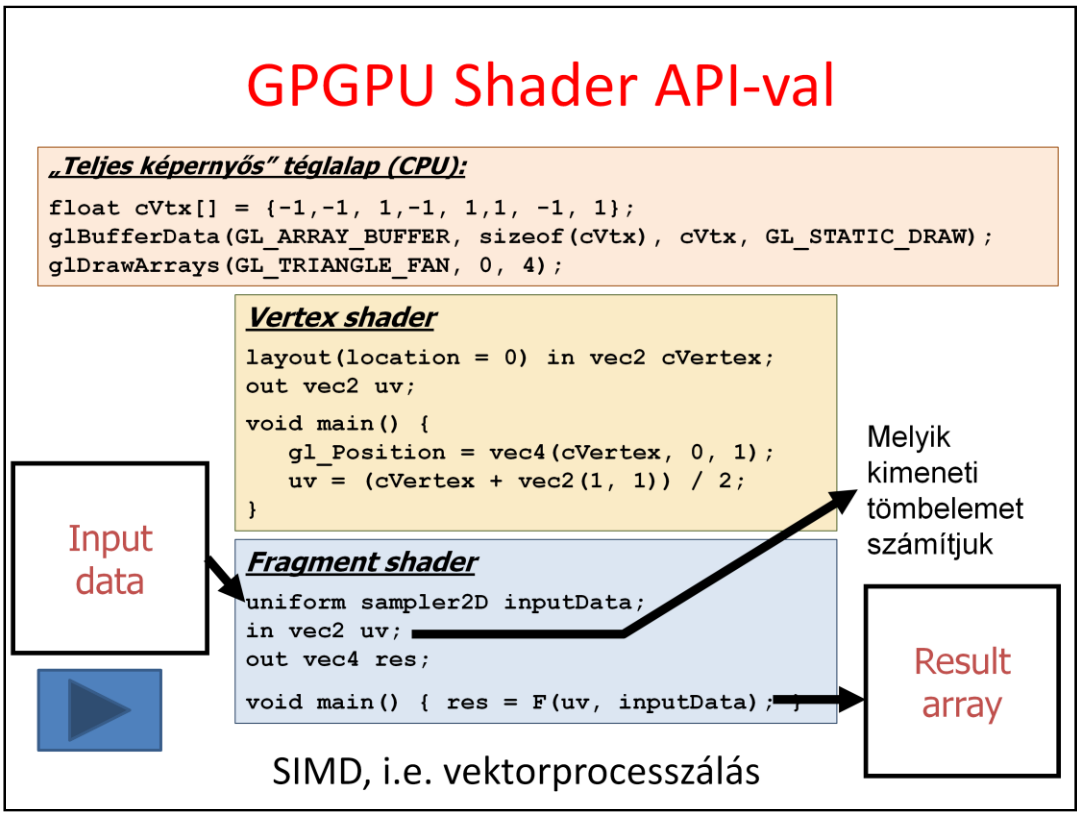
The amount of distortion can be computed exploiting the equivalence principle that states that gravity and acceleration are the same and cannot be distinguished. We can easily determine the amount of light bending when our room is accelerating in a direction. Then, the same formula is applied when the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

Hullám: Wave



```
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.



GPGPU Shader API-val

„Teljes képernyős" téglalap (CPU):

```
float cVtx[] = {-1,-1, 1,-1, 1,1, -1, 1};
glBufferData(GL_ARRAY_BUFFER, sizeof(cVtx), cVtx, GL_STATIC_DRAW);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

Vertex shader

```
layout(location = 0) in vec2 cVertex;
out vec2 uv;

void main() {
   gl_Position = vec4(cVertex, 0, 1);
   uv = (cVertex + vec2(1, 1)) / 2;
}
```

Fragment shader

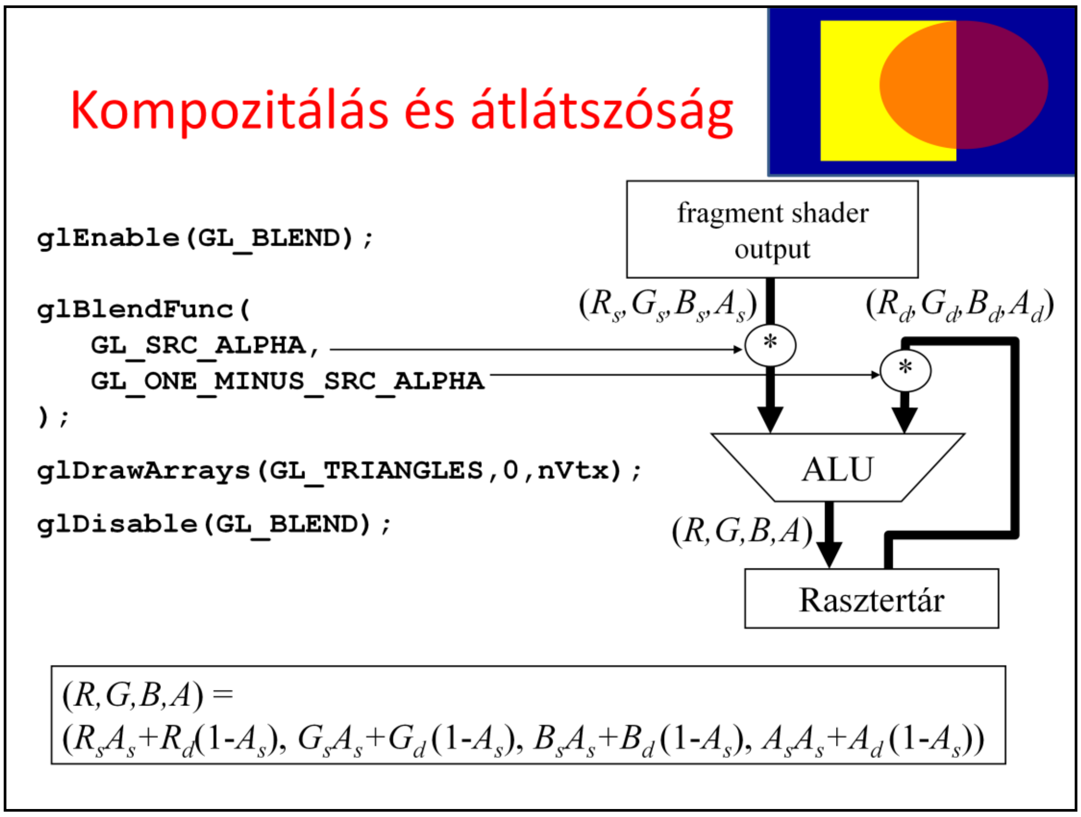
```
uniform sampler2D inputData;
in vec2 uv;
out vec4 res;

void main() { res = F(uv, inputData); }
```

SIMD, i.e. vektorprocesszálás

In a general GPGPU application the CPU renders a full viewport quad where vertices are directly specified in normalized device space.

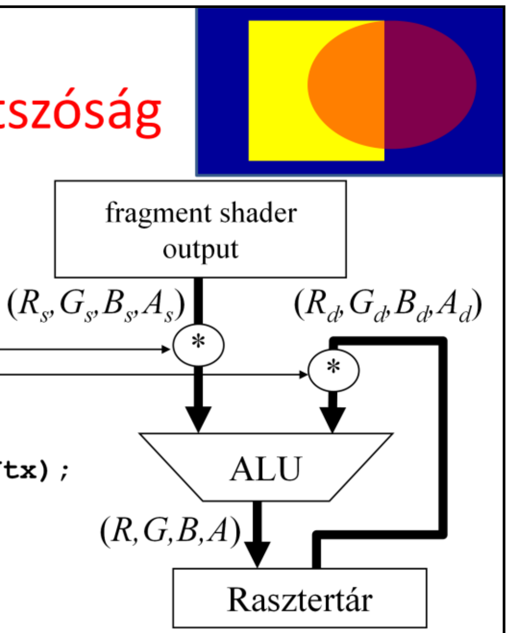
The vertex shader copies the vertex without transformation since it is already in normalized device space and the texture coordinate associated with this quad.

The fragment shader gets the interpolated texture coordinate which tells the shader which output element it computes and thus different fragment shaders would use different input data based on this (it would not make sense to compute the same result many times). The fragment shader can implement any function F that is based in the Input data and also on the Texture coordinate identifying the output index.



Kompozitálás és átlátszóság

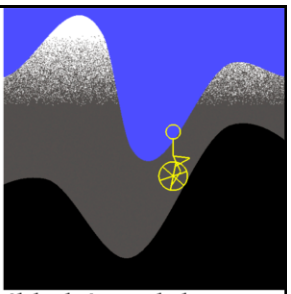
```
glEnable(GL_BLEND);

glBlendFunc(
    GL_SRC_ALPHA,
    GL_ONE_MINUS_SRC_ALPHA
);

glDrawArrays(GL_TRIANGLES,0,nVtx);

glDisable(GL_BLEND);
```



$$(R,G,B,A) = (R_sA_s+R_d(1\text{-}A_s),\ G_sA_s+G_d(1\text{-}A_s),\ B_sA_s+B_d(1\text{-}A_s),\ A_sA_s+A_d(1\text{-}A_s))$$

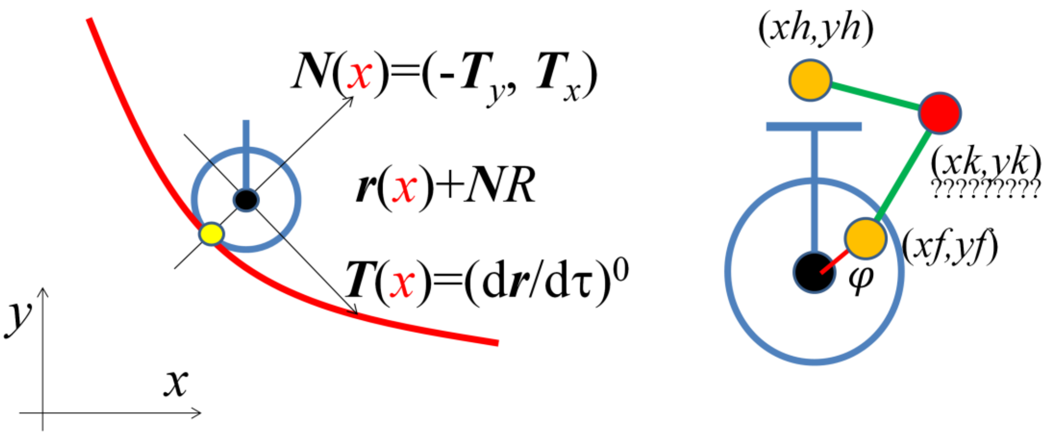
Between the output of the fragment shader and the raster memory there is a final hardware unit that is responsible for merging the new color values. The simplest case is rewriting the old color by a new one. With this merging unit, other combinations are also possible. The merging unit is enabled with glEnable(GL_BLEND) and disabled with glDisable. glBlendFunc selects the current type of composition. **glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**) is particularly popular since it simulates transparent color where transparency is the complementer of the opacity defined by the alpha channel or the color.

1. házi: Monocycle



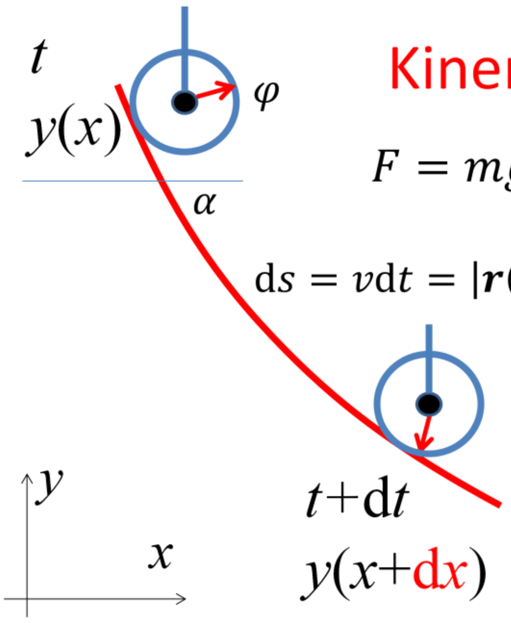
Írjon 2D **egykerekű bicikli szimulátor** programot.
A pálya negatív tenziójú explicit(!) Kochanek-Bartels spline, amelynek kontrollpontjait az egér bal gombjának lenyomásával lehet megadni, tetszőleges időpontban és sorrendben. A bicikli vonalas ábra, forgó, többküllős kerékkel és vadul pedálózó biciklissel. A kamera ortogonális vetítéssel dolgozik, kezdetben rögzített, majd SPACE hatására a biciklizőt követi. A biciklis a kezdeti kamera két széle között teker fel és alá, amit kb. 5 másodperc alatt tesz meg. A biciklis állandó erővel teker, amivel a nehézségi erőt és a sebességgel arányos légellenállást küzdi le. A tömeg az izmok erejéhez képest elhanyagolható, így a sebesség gyorsan változhat.
A távoli háttér egy procedurálisan textúrázott téglalap, ami az égboltot és pozitív tenziójú explicit(!) Kochanek-Bartels spline-nal definiált hegységet ábrázol. A távoli háttér nem követi a kamerát a hamis perspektíva érdekében.
Extrák: Az út mellett tereptárgyak helyezhetők el. A biciklis orientációja fizikailag helyesen úgy változik, hogy a forgatónyomaték zérus legyen.

Bicikli transzformációja



$$y(x)=a_3(x-x_i)^3+a_2(x-x_i)^2+a_1(x-x_i)+a_0$$
$$y'(x)=3a_3(x-x_i)^2+2a_2(x-x_i)+a_1$$

Kinematika



$$F = mg \cdot \sin(\alpha) + \rho v$$

$$\mathrm{d}s = v\mathrm{d}t = |\boldsymbol{r}(x + \mathrm{d}x) - \boldsymbol{r}(x)| = \left|\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}x}\right| \mathrm{d}x$$

$= \sqrt{1 + (y')^2}$

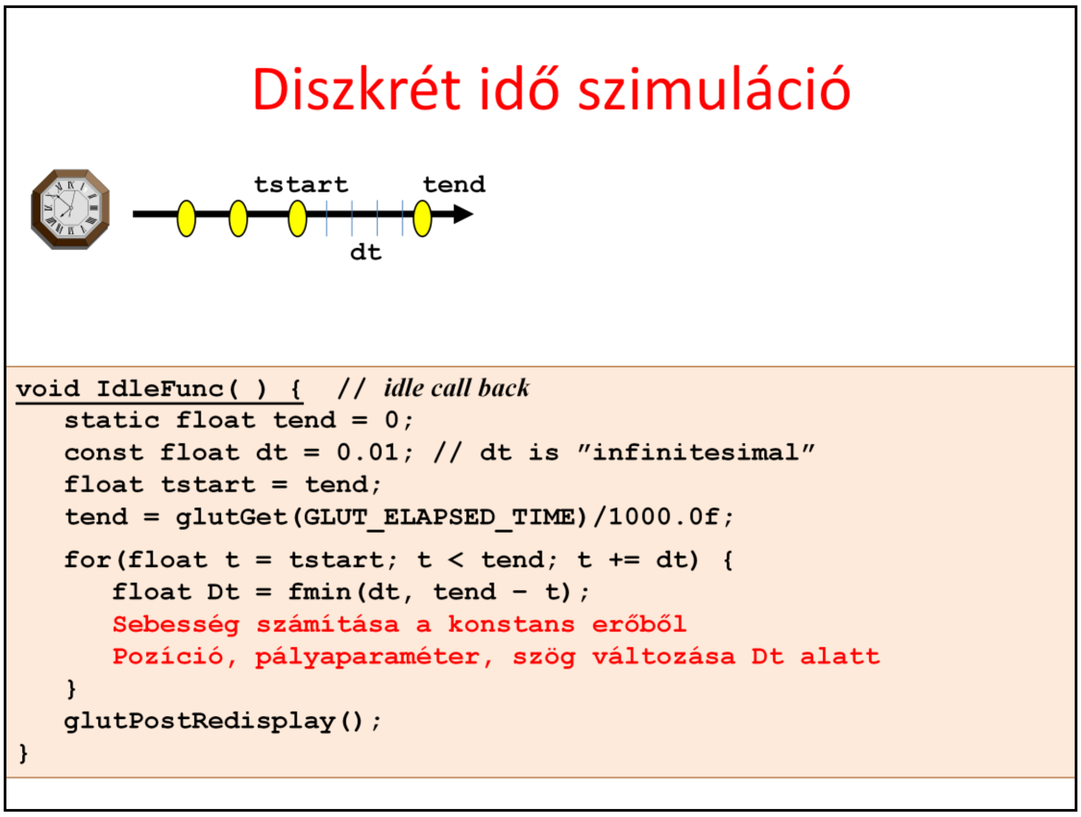
Állapot: $x, \varphi \rightarrow y, y', \alpha$

$$v = \frac{F - mg \cdot \sin(\alpha)}{\rho}$$

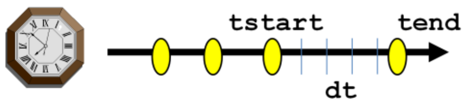
$$\mathrm{d}s = v\mathrm{d}t$$

$$\mathrm{d}x = \mathrm{d}s/\sqrt{1 + (y')^2}$$

$$\mathrm{d}\varphi = \mathrm{d}s/R$$



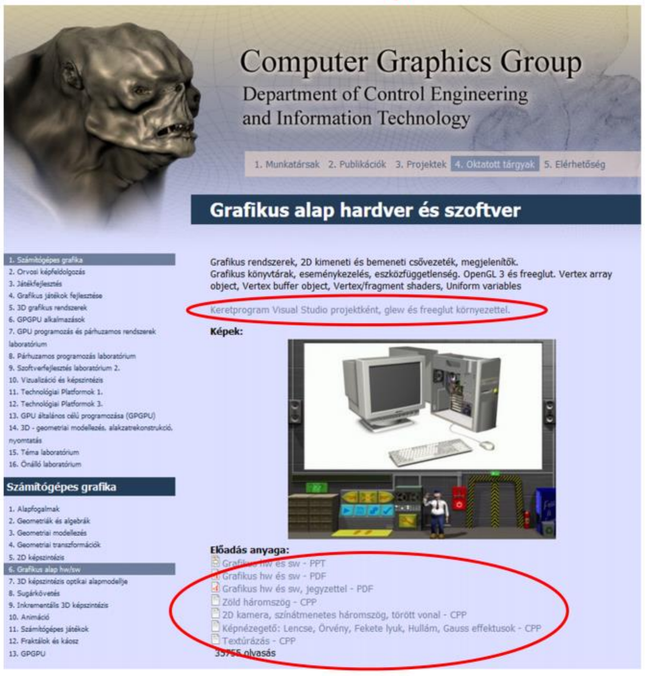
Diszkrét idő szimuláció



```
void IdleFunc( ) {   // idle call back
    static float tend = 0;
    const float dt = 0.01; // dt is "infinitesimal"
    float tstart = tend;
    tend = glutGet(GLUT_ELAPSED_TIME)/1000.0f;
    for(float t = tstart; t < tend; t += dt) {
        float Dt = fmin(dt, tend - t);
        Sebesség számítása a konstans erőből
        Pozíció, pályaparaméter, szög változása Dt alatt
    }
    glutPostRedisplay();
}
```

Animation also means that when some time elapses, the state of the virtual world catches up with the elapsed time. This should be happening even if the user does not even touch the computer. The event handling system provides the idle callback for this purpose. So in an idle call back, the elapsed times is computed, and the time interval [tstart,tend] elapsed since the last idle callback is simulated. As there is no upper limit for the length of the simulated interval, it is decomposed to dt steps that are sufficiently small. Here small means that time differentials can be well approximated differences and in dt we can suppose that the velocity and acceleration are constants.

Házi keret könyvtárakkal



Computer Graphics Group
Department of Control Engineering and Information Technology

1. Munkatársak 2. Publikációk 3. Projektek 4. Oktatott tárgyak 5. Elérhetőség

Grafikus alap hardver és szoftver

1. Számítógépes grafika
2. Orvosi képfeldolgozás
3. Játékfejlesztés
4. Grafikus játékok fejlesztése
5. 3D grafikus rendszerek
6. GPGPU alkalmazások
7. GPU programozás és párhuzamos rendszerek laboratórium
8. Párhuzamos programozás laboratórium
9. Szoftverfejlesztés laboratórium 2.
10. Vizualizáció és képszintézis
11. Technológiai Platformok 1.
12. Technológiai Platformok 3.
13. GPU általános célú programozása (GPGPU)
14. 3D - geometriai modellezés, alakzatrekonstrukció, nyomtatás
15. Téma laboratórium
16. Önálló laboratórium

Számítógépes grafika

1. Alapfogalmak
2. Geometriák és algebrák
3. Geometriai modellezés
4. Geometriai transzformációk
5. 2D képszintézis
6. Grafikus alap hw/sw
7. 3D képszintézis optikai alapmodellje
8. Sugárkövetés
9. Inkrementális 3D képszintézis
10. Animáció
11. Számítógépes játékok
12. Fraktálok és káosz
13. GPGPU

Grafikus rendszerek, 2D kimeneti és bemeneti csővezeték, megjelenítők.
Grafikus könyvtárak, eseménykezelés, eszközfüggetlenség. OpenGL 3 és freeglut. Vertex array object, Vertex buffer object, Vertex/fragment shaders, Uniform variables

Keretprogram Visual Studio projektként, glew és freeglut környezettel.

Képek:

Előadás anyaga:
- Grafikus hw és sw - PPT
- Grafikus hw és sw - PDF
- Grafikus hw és sw, jegyzettel - PDF
- Zöld háromszög - CPP
- 2D kamera, színátmenetes háromszög, törött vonal - CPP
- Képnézegető: Lencse, Örvény, Fekete lyuk, Hullám, Gauss effektusok - CPP
- Textúrázás - CPP

35755 olvasás