WebGl

<https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial>

<https://developer.mozilla.org/pl/docs/Web/API/WebGL_API/Tutorial>

**WebGL (Web-based Graphics Language) is a technology** that lets us create 2D and 3D graphics within a web browser.

The first prototype of WebGL was published at 2006.

It uses HTML5 canvas element and DOM interface.

This is done by using a JavaScript API that interacts with the GPU (Graphics Processing Unit) without the use of any plug-ins.

WebGL elements can be mixed with other HTML elements.

WebGL programs consist of control code written in JavaScript and shader code that is executed on a computer's GPU.

WebGL is designed and developed by the non-profit Khronos Group.

The Khronos Group is the non-profit organization founded in 2000 by companies ATI Technologies, Intel Corporation, NVIDIA, Silicon Graphics, Sun Microsystems and others.

We can check if our browser supports standard for example at:

get.webgl.org/

akirodic.com/p/jellyfish/

* Open Graphics Library (OpenGL) is a cross-language, cross-platform API for rendering 2D and 3D vector graphics.
* The API is used to interact with a GPU, to achieve hardware-accelerated rendering.
* Silicon Graphics started developing OpenGL in 1991 and released it in January 1992.

OpenGL is also used for parallel computing.  
OpenGL works in client-server architecture.

The client is an application that uses OpenGL. The server is an implementation of OpenGL, for example in the video card driver.   
OpenGL is a state machine (finite state machine). The state machine has an impact on rendering the object.

The feature set is made up of 250 basic requests, enabling the building of complex three-dimensional scenes of the basic geometric figures.  
  
API (Application Programming Interface).

**Getting Context**

When we draw inside of a canvas element, we have options of how we produce image.

Options (context of the canvas) corresponds to a different APIs with different available functionality and implementation details.

At the moment there are two canvas contexts: "2D" and "webgl".

To obtain a context, we use the canvas method getContext().

This method takes a context name as a first parameter and an optional second argument (browser dependent).

1. **WebGL (Web Graphics Library) is a JavaScript API** for rendering high-performance interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins.
2. **WebGL does so** by introducing an API that closely conforms to **OpenGL ES 2.0** that can be used in HTML5 <canvas> elements.
3. **This conformance makes it possible** for the API to take advantage of hardware graphics acceleration provided by the user's device.
4. **Support for WebGL is present** in most modern browsers however, the user's device must also have hardware that supports these features.
5. **Open Graphics Library (OpenGL) is** a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.
6. **The API** (OpenGL) is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.
7. **Silicon Graphics Inc.**, began developing OpenGL in 1991 and released it on June 30, 1992; applications use it extensively in the fields of:
   1. computer-aided design (CAD),
   2. virtual reality,
   3. scientific visualization,
   4. information visualization,
   5. flight simulation,
   6. video games.
8. **Since 2006**, OpenGL has been managed by the non-profit technology consortium Khronos Group.
9. **OpenGL for Embedded Systems (OpenGL ES)** is a subset of the OpenGL computer graphics rendering application programming interface (API) for rendering 2D and 3D computer graphics such as those used by:
   1. video games,
   2. typically hardware-accelerated using a graphics processing unit (GPU).
10. It is designed for embedded systems like **smartphones, tablet, video game consoles** etc.
11. OpenGL ES is the "most widely deployed 3D graphics API in history".
12. **OpenGL Shading Language (GLSL),**
    1. Is a high-level shading language with a syntax based on the C programming language.
    2. It was created by the OpenGL ARB (OpenGL Architecture Review Board).
    3. Now the GLSL specification control belongs to the Khronos Group.
    4. GLSL give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware-specific languages.
    5. ARB assembly language is a low-level shading language, which can be characterized as an assembly language. It was created by the OpenGL ARB to standardize GPU instructions controlling the hardware graphics pipeline.
13. **Graphics pipeline**
    1. In computer graphics, a computer graphics pipeline, rendering pipeline or simply graphics pipeline, is a conceptual model that describes what steps a graphics system needs to perform to render a 3D scene to a 2D screen.
    2. Once a 3D model has been created, for instance in a video game or any other 3D computer animation, the graphics pipeline is the process of turning that 3D model into what the computer displays.
    3. Because the steps required for this operation depend on the software and hardware used and the desired display characteristics, there is no universal graphics pipeline suitable for all cases.
    4. However, graphics application programming interfaces (APIs) such as Direct3D and OpenGL were created to unify similar steps and to control the graphics pipeline of a given hardware accelerator.
    5. These APIs abstract the underlying hardware and keep the programmer away from writing code to manipulate the graphics hardware accelerators (AMD/Intel/NVIDIA etc.).
    6. The model of the graphics pipeline is usually used in real-time rendering. Often, most of the pipeline steps are implemented in hardware, which allows for special optimizations.
    7. The term "pipeline" is used in a similar sense to the pipeline in processors: the individual steps of the pipeline run parallel but are blocked until the slowest step has been completed.

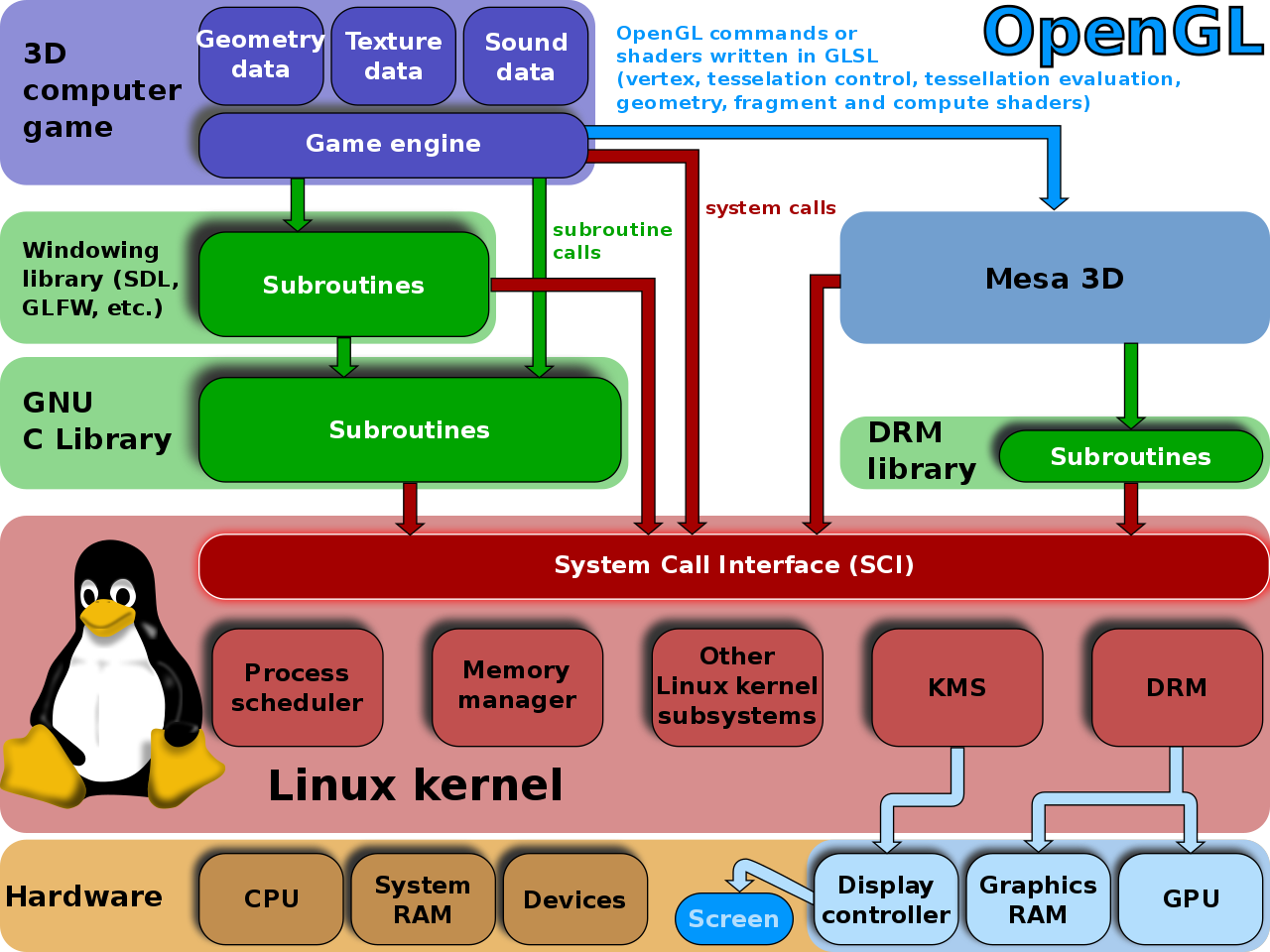
WebGL enables web content to use an API based on OpenGL ES 2.0 to perform 2D and 3D rendering in an HTML canvas in browsers that support it without the use of plug-ins.

WebGL programs consist of control code written in:

* JavaScript and
* shader code (GLSL) that is executed on a computer's Graphics Processing Unit (GPU).

WebGL elements can be mixed with other HTML elements and composited with other parts of the page or page background.

There are frameworks available that encapsulate WebGL's capabilities, making it easier to build 3D applications and games, such as THREE.js.



**The API is cross-language and multi-platform.**

* Mesa, also called **Mesa3D** and The Mesa 3D Graphics Library, is an open source software implementation of OpenGL and other graphics API specifications. Mesa translates these specifications to vendor-specific graphics hardware drivers.
* Simple DirectMedia Layer (**SDL**) is a cross-platform software development library designed to provide a hardware abstraction layer for computer multimedia hardware components. Software developers can use it to write high-performance computer games and other multimedia applications that can run on many operating systems such as Android, iOS, Linux, macOS, and Window
* GLFW is a lightweight utility C library for use with OpenGL. GLFW stands for Graphics Library Framework. It provides programmers with the ability to create and manage windows and OpenGL contexts, as well as handle joystick, keyboard and mouse input.

**WebGL – 3D graphics**

**Meshes, Polygons, and Vertices**

While there are several ways to draw 3D graphics, by far the most common is to use a mesh.

A mesh is an object composed of one or more polygonal shapes, constructed out of vertices (x, y, z) defining coordinate positions in 3D space.

The polygons most typically used in meshes are triangles and quads (groups of four vertices).

3D meshes are often referred to as models.



Figure 1-3. A 3D mesh

**Materials, Textures, and Lights**

The surface of a mesh is defined using additional attributes beyond the x, y, z vertex positions.

Surface attributes can be as simple as a single solid color, or they can be complex, comprising several pieces of information that define, for example, how light reflects off the object or how shiny the object looks.

Surface information can also be represented using one or more bitmaps, known as texture maps (textures).

In graphics systems, the surface properties of a mesh are referred to collectively as materials.

Materials typically rely on the presence of one or more lights, which define how a scene is illuminated.

**Transforms and Matrices**

3D meshes are defined by the positions of their vertices.

Most 3D systems support transforms, operations that move the whole mesh not each vertex separately.

Transforms allow a rendered mesh to be scaled, rotated, and moved around, without actually changing any values in its vertices.

A transform is typically represented by a matrix containing an array of values used to compute the transformed positions of vertices.

**Cameras, Perspective, Viewports, and Projections**

Camera – an object that defines where (relative to the scene) the user is positioned and oriented, as well as other real-world camera properties such as the size of the field of view, which defines perspective (i.e., objects farther away appearing smaller).

The camera’s properties combine to deliver the final rendered image of a 3D scene into a 2D viewport defined by the window or canvas.

Cameras are almost always represented using a couple of matrices.

The first matrix defines the position and orientation of the camera.

The second matrix is called the projection matrix.



Figure 1-4. Camera, viewport, and projection

**Shaders**

To render the final image for a mesh, a developer must define exactly how vertices, transforms, materials, lights, and the camera interact with one another to create that image.

This is done using shaders.

A shader (also known as a programmable shader) is a chunk of program code that implements algorithms to get the pixels for a mesh onto the screen.

Shaders are typically defined in a high-level C-like language and compiled into code usable by the graphics processing unit (GPU).

WebGL requires shaders.

Popular libraries written for WebGL come with prebuilt shaders.

**WebGL Components**

**The Drawing Buffers**

WebGL has a color buffer, depth buffer and stencil buffer.

A buffer is a block of memory that can be written to and read from and temporarily stores data.

The **color buffer** holds color information (RGB) values and optionally an alpha value that stores the amount of transparency.

The **depth buffer** stores information on a pixel’s depth component (z-value).

As the map from 3D world space to 2D screen space can result in several points being projected to the same (x,y) canvas value, the z-values are compared and only one point, usually the nearest, is kept and rendered.

The **stencil buffer** is used to outline areas to render or not render.

When an area of an image is marked off to not render, it is known as masking that area.

The entire image, including the masked portions, is known as a stencil.

**Primitive Types**

Primitives are the graphical building blocks that all models in a particular graphics language are built with.

In WebGL, there are three primitive types:

* points,
* lines,
* triangles.

There are seven ways to primitive types:

* POINTS,
* LINES,
* LINE\_STRIP,
* LINE\_LOOP,
* TRIANGLES,
* TRIANGLE\_STRIP,
* TRIANGLE\_FAN.



Figure 1-3.

**POINTS** are vertices (spatial coordinates) rendered one at a time.

**LINES** are formed along pairs of vertices.

**LINE\_STRIP** is a collection of vertices in which, except for the first line, the starting point of each line is the end point of the previous line.

**LINE\_LOOP** is similar to LINE\_STRIP except that it is a closed off loop with the last vertex connecting back to the very first.

**TRIANGLES** are made from three vertex.

**TRIANGLE\_STRIP** uses the last two vertices along with the next vertex to form triangles.

**TRIANGLE\_FAN** uses the first vertex specified as part of each triangle.

**Vertex Data**

WebGL does not have fixed functionality it uses programmable shaders.

We cannot directly set the color or location of a vertex directly into a scene.

All data associated with a vertex needs to be streamed (passed along) from the JavaScript API to the GPU.

With WebGL, we have to create **vertex buffer objects (VBOs**) that will hold vertex attributes such as position, color, normal, and texture coordinates.

These vertex buffers are then sent to a shader program that can use and manipulate the passed-in data.

Using shaders instead of having fixed functionality is central to WebGL.

**Vertex Buffer Objects (VBOs)**

Each VBO stores data about a particular attribute of vertices.

This could be position, color, a normal vector, texture coordinates, or something else.

To create a buffer, we call:

WebGLBuffer createBuffer()

and store the returned object, like so:

var myBuffer = gl.createBuffer();

Next we bind the buffer using:

void bindBuffer(GLenum target, WebGLBuffer buffer)

like this:

gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, myBuffer);

The target parameter is either

gl.ARRAY\_BUFFER

or

gl.ELEMENT\_ARRAY\_BUFFER.

The target ELEMENT\_ARRAY\_BUFFER is used when the buffer contains vertex indices, and ARRAY\_BUFFER is used for vertex attributes such as position and color.

Once a buffer is bound and the type is set, we can place data into it with this function:

void bufferData(GLenum target, ArrayBuffer data, GLenum usage)

The usage parameter of the bufferData call can be one of:

STATIC\_DRAW,

DYNAMIC\_DRAW,

STREAM\_DRAW.

STATIC\_DRAW will set the data once and never change throughout the application’s use of it, which will be many times.

DYNAMIC\_DRAW will also use the data many times in the application but will respecify the contents to be used each time.

STREAM\_DRAW is similar to STATIC\_DRAW in never changing the data, but it will be used at most a few times by the application.

Using this function looks like the following:

var data = [ 1.0, 0.0, 0.0,

0.0, 1.0, 0.0,

0.0, 1.0, 1.0

];

gl.bufferData(gl.ARRAY\_BUFFER, data, gl.STATIC\_DRAW);

Altogether the procedure of creating, binding and storing data inside of a buffer looks like:

var data = [ 1.0, 0.0, 0.0,

0.0, 1.0, 0.0,

0.0, 1.0, 1.0

];

var myBuffer = gl.createBuffer();

gl.bindBuffer(gl.ARRAY\_BUFFER, myBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, data, STATIC\_DRAW);

Notice that in the gl.bufferData line, we do not explicitly specify the buffer to place the data into.

WebGL implicitly uses the currently bound buffer.

We can delete a buffer with a call to this:

void deleteBuffer(WebGLBuffer buffer);

**Attributes and Uniforms**

Vertices have **attributes** which can be passed to shaders.

We can also pass **uniform** values to the shader which will be constant for each vertex.

As the shader is a compiled external program, we need to be able to reference the location of all variables within the program.

**Variable Qualifiers**

Qualifiers give a special meaning to the variable. The following qualifiers are available:

* **const** – The declaration is of a compile time constant.
* **attribute** – Global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders. This qualifier can only be used in vertex shaders. For the shader this is a read-only variable.
* **uniform** – Global variables that may change per primitive [...], that are passed from the OpenGL application to the shaders. This qualifier can be used in both vertex and fragment shaders. For the shaders this is a read-only variable.
* **varying** – used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.

Once we obtain the location of a variable, we can send data to the shader from our web application.

To get the location of an attribute or uniform within the WebGL program, we use these API calls:

GLint getAttribLocation(WebGLProgram program, DOMString name)

WebGLUniformLocation getUniformLocation(WebGLProgram program, DOMString name)

The GLint and WebGLUniformLocation return values are references to the location of the attribute or uniform

within the shader program.

The first parameter is our WebGLProgram object and the second parameter is the attribute name as found in the vertex or fragment shader source.

If we have an attribute in a shader by the name of "aVertexPosition", we obtain its position within our JavaScript like this:

var vertexPositionAttribute = gl.getAttribLocation(glProgram, "aVertexPosition");

If we are sending an array of data to an attribute, we have to enable array data with a call to this:

void enableVertexAttribArray(GLuint index)

Here, the index is the attribute location that we previously obtained and stored. The return value is void because the function returns no value.

With our previously defined attribute location, this call looks like the following:

gl.enableVertexAttribArray(vertexPositionAttribute);

Now that we have the location of an attribute and have told our shader that we will be using an array of values, we assign the currently bound ARRAY\_BUFFER target to this vertex attribute as we have demonstrated in the previous section:

gl.bindBuffer(gl.ARRAY\_BUFFER, myBuffer);

Finally, we let our shader know how to interpret our data.

We need to remember that the shader knows nothing about the incoming data.

Just because we name an array to help us understand what data it contains, such as myColorData, the shader just sees data without any context.

The API call to explain our data format is as follows:

void vertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei

stride, GLintptr offset)

size is the number of components per attribute.

For example, with RGB colors, it would be 3; and with an alpha channel, RGBA, it would be 4.

If we have location data with (x,y,z) attributes, it would be 3; and if we had a fourth parameter w, (x,y,z,w), it would be 4.