WebGL

Hilding Lindén; 39151

Kandidatavhandling i Datavetenskap

Handledare: Jan Westerholm

Fakulteten för naturvetenskaper och teknik

Åbo Akademi

2018

Innehåll

1 Inledning

En stor del av den interaktiva underhållningen som finns idag består av dator- och mobilspel. Internet växer snabbt och det finns många spel som kan spelas via webbläsare både på stationära datorer och mobila enheter. Orsaken till detta är att de flesta har tillgång till internet vilket betyder att spelutvecklare kan nå så många kunder som möjligt. Tidigare har Adobe Flash Player, också kallat Flash, använts för att skapa dessa webbläsarspel men nyligen har det nya programmeringsgränssnittet WebGL blivit mer vanligt. För att spela spel som är programmerade i Flash måste användaren installera programmet som ett insticksprogram i webbläsare. Eftersom Flash som ett insticksprogram är svårare att använda och mindre säkert än integrerad funktionalitet väljer många utvecklare att använda WebGL istället.

I denna avhandling kommer jag att förklara vad WebGL är och var det kommer ifrån. Jag kommer också att berätta hur en enkel WebGL applikation är uppbyggd och ge exempel på hur den kunde se ut.

2 Bakgrund

WebGL (Web Graphics Library) är ett programmeringsgränssnitt baserat på skriptspråket JavaScript. WebGL används för att rendera interaktiv 2D- och 3D-grafik i webbläsare och är fullständigt integrerad i alla webbstandarder, bland annat rekommendationerna av World Wide Web Consortium (W3C).

2D- och 3D-grafik, som fungerar med hjälp av insticksprogram, i webbläsaren renderas traditionellt av processorn men i och med att WebGL är integrerat i webbläsaren renderas allt på grafikkortet. Då grafiken kan renderas på grafikkortet minskar inmatnings fördröjningen och tiden det tar för en scen att renderas.

De flesta företagen har idag en eller flera webbsidor med information och tjänster för kunder. För att göra webbsidor lättare att navigera och visuellt mer tilltalande kan 3D-gränssnitt som WebGL användas.

3 OpenGL / OpenGL ES

WebGL är baserat på OpenGL specifikationen. OpenGL är endast en specifikation vilket betyder att grafikkortstillverkarna är ansvariga för själva implementeringen av gränssnittet på hårdvaran.

OpenGL var ursprungligen utvecklat av Silicon Graphics Inc. och publicerades som version 1.0 år 1992. Efter år 2006 har Khronos Group stått för utvecklingen av OpenGL och den nyaste versionen av OpenGL är 4.6 och släpptes 2017.

Andra programmeringsgränssnitt för grafikprogrammering är Microsofts Direct3D 11 och Direct3D 12 men som fungerar endast på Windows operativ system, Metal som fungerar endast på iOS och macOS och plattformsoberoende Vulkan som är baserat på Mantle av AMD.

OpenGL for Embedded Systems (OpenGL ES eller GLES) är en delmängd av OpenGL specifikationerna som är avsedd för inbyggda system som till exempel smarttelefoner och pekplattor. OpenGL ES 1.0 publicerades år 2003 och version 2.0, som WebGL är baserat på släptes år 2007.

4 GLSL

OpenGL Shading Language, förkortat GLSL, är ett funktionellt programmeringsspråk som används för att programmera shaders i OpenGL.

GLSL är baserat på programmeringsspråket C och har alla operatörer som finns i C och C++, förutom pekare. Rekursion är inte möjlig i GLSL.

GLSL ES är den delmängden av GLSL som används för shaderprogrammering i OpenGL ES och som också används i WebGL.

5 WebGL

De flesta webbläsarna på Windows operativsystemen använder ANGLE (Almost Native Graphics Layer Engine) för att översätta OpenGL ES-metodanropen till Direct3D-anrop för bättre drivrutinsstöd. ANGLE översätter även GLSL till

HLSL (High-Level Shading Language) som är programmeringsspråket för shaders i Direct3D gränssnittet.

6 Innehållet av en WebGL applikation

I detta avsnitt går jag igenom den grundläggande strukturen av en mycket enkel WebGL applikation. Källkoden för en exempelapplikation finns i bilaga A och B.

En WebGL applikation består i grund och botten av 3 delar: HTML5, JavaScript och GLSL. HTML5 och JavaScript exekveras på processorn medan GLSL exekveras på grafikprocessorn.

<canvas>

<canvas> är ett element i HTML5 som används för att rita grafik på en webbsida med hjälp av JavaScript. För att kunna använda WebGL måste ett WebGL renderingskontext fås från elementet i HTML5. Detta görs i genom att använda metoden getElementById(), med namnet av canvaselementet som parameter, för att få ett canvasobjekt. getContext() anropas sedan på canvasobjektet för att få renderingsobjektet.

Vertexbuffert

Tredimensionella objekt kan beskrivas som en samling av tvådimensionella geometriska figurer kallade polygoner. Polygonerna består av sidor och hörn där dessa sidors ändpunkter möts. Inom datorgrafik heter polygonhörnen vertex.

I OpenGL och WebGL sparas vertices i en minnesbuffert som heter vertexbuffert. Varje vertex i bufferten kan innehålla information om bland annat hörnets position, normalvektor och färg. Den informationen som sparas i bufferten är oftast av typen ”float” vilket innebär att det handlar om ett flyttal, eller mer allmänt känt som decimaltal, som är 32 bitar stort.

I WebGL anges positionen av en vertex som ett flyttal mellan -1.0 och 1.0 i både x- och y-axeln och börjar från det nedre hörnet till vänster (-1.0, -1.0). Origo (positionen 0.0, 0.0) av koordinatsystemet är i mitten av canvaselementet.

OpenGL fungerar som en automat, det vill säga programmeraren måste i förväg ange bland annat vilken buffert som ska användas och vilka shaders som ska användas. I praktiken binds buffertar fast i buffertobjekt och får information före exekvering.

För att definiera geometrin av objektet som ska ritas skapas först ett buffertobjekt med metoden createBuffer(), sedan binds en räcka fast i buffertobjektet med metoden bindBuffer() som sedan får information om geometrin med metoden bufferData().

Shader

En shader är den kod eller program som exekveras på grafikprocessorn. I OpenGL används språket OpenGL Shading Language (GLSL) som är baserat på programmeringsspråket C. I datorgrafik delas shaders upp i två olika typer: vertexshader och fragment- eller pixelshader.

En vertexshader är som namnet säger en shader som har hand om polygonsidorna av en figur och körs en gång per vertex. Denna shader används oftast för att räkna ut förändring i positionen av en vertex till exempel som följd av perspektivändring eller kamerarörelse.

En fragmentshader eller pixel shader har istället hand om varje pixel eller bildpunkt i en figur och körs därför mycket oftare än en vertexshader. Fragmentshadern används för att räkna ut färgen av varje pixel utgående från informationen i vertexbufferten. Denna shader räknar också ut färgen beroende på omgivningen genom att implementera till exempel strålföljning (eng. ray tracing).

Nästa steg i applikationen är att ange och fästa shaderprogrammen. Lika som med vertexbufferten måste ett objekt skapas. Detta görs med metoden createShader(), sedan anges vilken shaderprogramkällkod som ska användas med metoden shaderSource(). Källkoden är en sträng och kan definieras i JavaScript filen eller

finnas i en skild text fil. Till sist kompileras shaderprogrammet som finns i shaderobjektet med metoden compileShader().

Dessa tre steg måste göras för alla typer av shaderprogram som används i applikationen. I exempelapplikationen används en vertexshader och en fragmentshader.

För att få shaderprogrammen länkade till applikationen skapas ett shaderprogramobjekt med metoden createProgram(). Till objektet fästes sedan de kompilerade shaderprogrammen med metoden attachShader()**.** De fästa shaderprogrammen måste länkas ihop med metoden linkProgram() för att fungera ihop. Till sist anges att det shaderprogramobjektet som har skapats är det objektet som ska användas vid exekvering med metoden useProgram().

Uniform och attribut

Uniforms och attribut är sätt att få data till shaderprogrammen vid exekvering. Uniform används då informationen man vill skicka inte ändras och har oftast namnkonventionen u_ i variabelnamnet. Attribut används då informationen är dynamisk och följer namngivningskonventionen a_ i variabelnamnet.

I exempelprogrammet används ett attribut som har namnet a_Position. För att kunna skicka information till shadern måste först platsen på attributet hittas. Detta sker med metoden getAttribLocation(). Då platsen finns tillgänglig måste den bindas fast i bufferten med hjälp av metoden vertexAttribPointer() till sist aktiveras attributet med metoden enableVertexAttribArray() för att kunna användas.

Primitiver

Då man i slutet av en WebGL applikation vill rita ett objekt finns det 3 olika primitiver att välja mellan: punkt, linje och triangel. Då man ritar en punkt renderas varje vertex skilt och då man ritar en linje renderas en linje mellan varje par av vertices. Om man väljer att rita en triangel renderas varje grupp av 3 vertices som en triangel.

Med trianglar kan man bygga alla 2D- och 3D objekt eftersom 3 punkter är det minsta antalet punkter som behövs för att beskriva ett plan.

I slutet av exempelprogrammet anges först vilken färgs som rutan ska rensas med och det blir den nya bakgrundsfärgen. Detta görs med metoden clearColor() som tar emot en färg av typen rgba där r är värdet av rött, g är värdet av grönt, b är värdet av blått och a är värdet av opacitet. Därefter rensas rutan med metoden clear(). Till sist ritas objektet med metoden drawArrays() som använder informationen om geometrin som finns i vertexbufferten och som exekverar shaderprogrammen som angets.

Resultatet av exempelprogrammet finns i bilaga C.

7 WebGL prestanda

Forskning har visat att WebGL har mycket kortare exekveringstid per bildruta än Flash och HTML5 [4]. Hoetzlein visade i ett test med 10000 sprites att Flash hade en exekveringstid på 78ms per bildruta och HTML5 hade en exekveringstid på 43ms per bildruta på Firefox medan WebGL inte översteg 50ms innan 700000 sprites [4].

8 Källor

[1] Shirley P. Marschner S., Fundamentals of Computer Graphics 3rd ed. ISBN-13: 978-1568814698

[2] Matsuda K. Lea R., WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL. ISBN-13: 978-0321902924

[3] Atitayaporn Muennoi, Daranee Hormdee, D. Zhang, B. Zi, G. Cui, H. Ding, "3D Web-based HMI with WebGL Rendering Performance", MATEC Web of Conferences, vol. 77, pp. 09003, 2016, ISSN 2261-236X.

[4] Rama C. Hoetzlein, ”Graphics Performance in Rich Internet Applications”, IEEE Computer Graphics and Applications (Volume: 32, Issue: 5, Sept.-Oct. 2012).

[5] Hyowon Kim, Sunwoo Nam, Joonghun Park and Daehyun Ko, “Direct Canvas: Optimized WebGL Rendering Model”.

A. Bilaga A

WebGL Exempel.html

```
<!DOCTYPE html>
<html>
            <head>
                        <meta charset="utf-8" />
                        <title>Ett WebGL Exempel</title>
            </head>

            <body onload="main()">
                        <canvas id="c" width="400" height="400">
                        Använd en webbläsare som stöder HTML5
                        </canvas>

                        <script src="WebGL Exempel.js"></script>

            </body>
</html>
```

B. Bilaga B

WebGL Exempel.js

```
var VSHADER_SOURCE =

    'attribute vec4 a_Position;\n' +
    '\n' +
    'void main() {\n' +
    '   gl_Position = a_Position;\n' +
    '}\n';

var FSHADER_SOURCE =
    'void main() {\n' +
    '   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
    '}\n';

function main() {
    var canvas = document.getElementById('c');

    var gl = canvas.getContext("webgl");

    var vs = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vs, VSHADER_SOURCE);
    gl.compileShader(vs);

    var fs = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fs, FSHADER_SOURCE);
    gl.compileShader(fs);

    var program = gl.createProgram();

    gl.attachShader(program, vs);
    gl.attachShader(program, fs);
```

```
    gl.linkProgram(program);
    gl.validateProgram(program);

    gl.deleteShader(vs);
    gl.deleteShader(fs);

    gl.useProgram(program);

    var vertices = new Float32Array([
         0.0,  0.5,
        -0.5, -0.5,
         0.5, -0.5
    ]);
    var n = 3;

    var vb = gl.createBuffer();

    gl.bindBuffer(gl.ARRAY_BUFFER, vb);
    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

    var a_Position = gl.getAttribLocation(program, 'a_Position');
    gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(a_Position);

    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.drawArrays(gl.TRIANGLES, 0, n);
}
```

C. Bilaga C

Resultatet av WebGL applikationen.