

(<https://davrous.com/about>) (<http://twitter.com/davrous>)

(<http://facebook.com/davrous>) (<https://linkedin.com/in/davrous>)

(<https://soundcloud.com/david-rousset>) (<https://www.davrous.com/feed/>)



David Rousset (<https://www.davrous.com/>)

Tutorial part 2: learning how
to write a 3D soft engine from
scratch in C#, TS or JS –
drawing lines & triangles

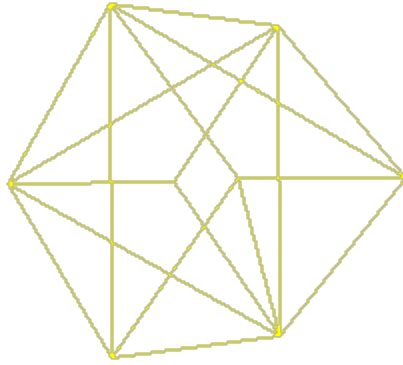
June 14, 2013 (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/>) / David Rousset (<https://www.davrous.com/author/davrous/>) / 3D Software Engine (https://www.davrous.com/category/3d-software-engine/?lang=en_us), English (https://www.davrous.com/category/english/?lang=en_us), Technical article (https://www.davrous.com/category/technical-article/?lang=en_us), Tutorial (https://www.davrous.com/category/tutorial/?lang=en_us)

Now that we have built the core of our 3D engine thanks to the previous tutorial Tutorial series- learning how to write a 3D soft engine from scratch in C#, TypeScript or JavaScript (<https://www.davrous.com/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/>), we can work on enhancing the rendering. The next step is then to connect the dots to draw some lines in order to render what you probably know as a **“wireframe” rendering**.

- 1 – Writing the core logic for camera, mesh & device object (<https://www.davrous.com/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/>)
- 2 – Drawing lines and triangles to obtain a wireframe rendering (this article)
- 3 – Loading meshes exported from Blender in a JSON format (<https://www.davrous.com/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/>)
- 4 – Filling the triangle with rasterization and using a Z-Buffer (<https://www.davrous.com/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-z-buffering/>)
- 4b – Bonus: using tips & parallelism to boost the performance (<https://www.davrous.com/2013/06/25/tutorial-part-4-bonus-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-optimizing-parallelism/>)
- 5 – Handling light with Flat Shading & Gouraud Shading (<https://www.davrous.com/2013/07/03/tutorial-part-5-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-flat-gouraud-shading/>)
- 6 – Applying textures, back-face culling and WebGL (<https://www.davrous.com/2013/07/18/tutorial-part-6-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-texture-mapping-back-face-culling-webgl/>)

In this tutorial, you will learn how to draw lines, what a face is and how cool is the Bresenham algorithm to draw some triangles.

Thanks to that, at the end, you will know how to code something as cool as that:



Yes! Our 3D rotating cube starts to really live on our screens!

First basic algorithm to draw a line between two points

Let's start by coding a simple algorithm. To **draw a line between 2 vertices**, we're going to use the following logic:

- if the distance between the 2 points (point0 & point1) is less than 2 pixels, there's nothing to do
- otherwise, we're finding the **middle point** between both points
(point0 coordinates + (point1 coordinates - point0 coordinates) / 2)
- we're drawing that point on screen
- we're launching this **algorithm recursively** between point0 & middle point and between middle point & point1

Here is the code to do that:

- C#
- TypeScript
- JavaScript

```

public void DrawLine(Vector2 point0, Vector2 point1)
{
    var dist = (point1 - point0).Length();

    // If the distance between the 2 points is less than 2 pixels
    // We're exiting
    if (dist < 2)
        return;

    // Find the middle point between first & second point
    Vector2 middlePoint = point0 + (point1 - point0)/2;
    // We draw this point on screen
    DrawPoint(middlePoint);
    // Recursive algorithm launched between first & middle point
    // and between middle & second point
    DrawLine(point0, middlePoint);
    DrawLine(middlePoint, point1);
}

```

```

public drawLine(point0: BABYLON.Vector2, point1: BABYLON.Vector2):
void {
    var dist = point1.subtract(point0).length();

    // If the distance between the 2 points is less than 2 pixels
    // We're exiting
    if (dist < 2)
        return;

    // Find the middle point between first & second point
    var middlePoint = point0.add((point1.subtract(point0)).scale(0
.5));
    // We draw this point on screen
    this.drawPoint(middlePoint);
    // Recursive algorithm launched between first & middle point
    // and between middle & second point
    this.drawLine(point0, middlePoint);
    this.drawLine(middlePoint, point1);
}

```

```

Device.prototype.drawLine = function (point0, point1) {
    var dist = point1.subtract(point0).length();

    // If the distance between the 2 points is less than 2 pixels
    // We're exiting
    if(dist < 2) {
        return;
    }

    // Find the middle point between first & second point
    var middlePoint = point0.add((point1.subtract(point0)).scale(0.5));
    // We draw this point on screen
    this.drawPoint(middlePoint);
    // Recursive algorithm launched between first & middle point
    // and between middle & second point
    this.drawLine(point0, middlePoint);
    this.drawLine(middlePoint, point1);
};

```

You need to update the rendering loop to use this new piece of code:

- C#
- TypeScript
- JavaScript

```

for (var i = 0; i < mesh.Vertices.Length - 1; i++)
{
    var point0 = Project(mesh.Vertices[i], transformMatrix);
    var point1 = Project(mesh.Vertices[i + 1], transformMatrix);
    DrawLine(point0, point1);
}

```

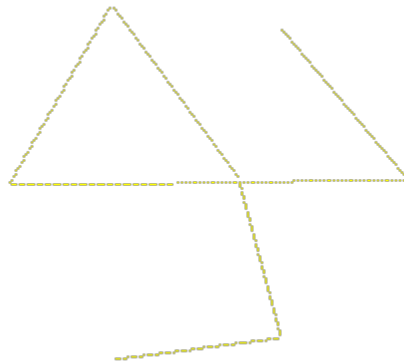
```

for (var i = 0; i < cMesh.Vertices.length -1; i++){
    var point0 = this.project(cMesh.Vertices[i], transformMatrix);
    var point1 = this.project(cMesh.Vertices[i + 1], transformMatrix);
    this.drawLine(point0, point1);
}

```

```
for (var i = 0; i < cMesh.Vertices.length - 1; i++){
    var point0 = this.project(cMesh.Vertices[i], transformMatrix);
    var point1 = this.project(cMesh.Vertices[i + 1], transformMatrix);
    this.drawLine(point0, point1);
}
```

And you should now obtain something like that:



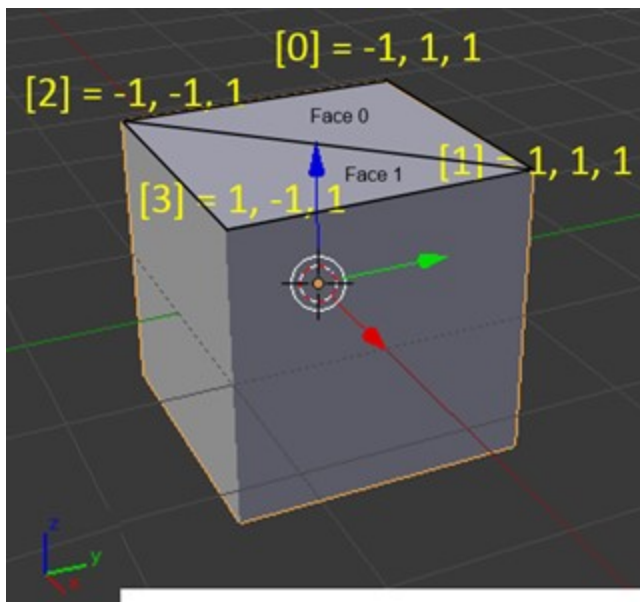
I know this looks weird but this was the expected behavior. It should help you starting to understand what you need to do to display a 3D mesh. But to have a better rendering, we need to discover a new concept.

Displaying faces with triangles

Now that we know how to draw lines, we need a better way to render the mesh with them. The **simplest geometric 2D shape is a triangle**. The idea in 3D is then to draw all our meshes by using those triangles. We then need to split each side of our cube into 2 triangles. We're going to do this "manually" but we'll see in the next tutorial that 3D modelers are doing this step automatically for us now.

To draw triangles, you need to have 3 points/vertices. A face is then simply a structure containing 3 values which are indexes pointing to the proper vertices array of the mesh to be rendered.

To be understand this concept, let's take our previous figure with a Cube displayed by Blender:



(https://msdnshared.blob.core.windows.net/media/MSDNBlogsFS/prod.evul.blogs.msdn.com/CommunityServer.Blogs.Components.WeblogFiles/00/00/01/10/46/metablogapi/7532.image_43428C9F.png)

We have 4 vertices displayed on this figure with the following indices: 0, 1, 2, 3. To draw the upper side of the cube, we need to draw 2 triangles. The first one, Face 0, will be drawn with 3 lines from vertex 0 (-1, 1, 1) to vertex 1 (1, 1, 1), from vertex 1 (1, 1, 1) to vertex 2 (-1, -1, 1) and finally from vertex 2 (-1, -1, 1) to vertex 0 (-1, 1, 1). The second triangle, Face 1, will be drawn with the lines from vertex 1 to vertex 2, vertex 2 to vertex 3 and vertex 3 to vertex 1.

The equivalent code would be something like that:

```
var mesh = new SoftEngine.Mesh("Square", 4, 2);
meshes.Add(mesh);
mesh.Vertices[0] = new Vector3(-1, 1, 1);
mesh.Vertices[1] = new Vector3(1, 1, 1);
mesh.Vertices[2] = new Vector3(-1, -1, 1);
mesh.Vertices[3] = new Vector3(1, -1, 1);

mesh.Faces[0] = new Face { A = 0, B = 1, C = 2 };
mesh.Faces[1] = new Face { A = 1, B = 2, C = 3 };
```

If you want to draw the whole cube, you need to find the 10 remaining faces as we've got **12 faces** for the 6 sides of our cube to draw.

Let's now define the code for a **Face** object. It's a very simple object as this is just a **set of 3 indexes**. Here's the code of Face and the new Mesh definition which also now use it:

- C#

- TypeScript
- JavaScript

```
namespace SoftEngine
{
    public struct Face
    {
        public int A;
        public int B;
        public int C;
    }
    public class Mesh
    {
        public string Name { get; set; }
        public Vector3[] Vertices { get; private set; }
        public Face[] Faces { get; set; }
        public Vector3 Position { get; set; }
        public Vector3 Rotation { get; set; }

        public Mesh(string name, int verticesCount, int facesCount
    )
        {
            Vertices = new Vector3[verticesCount];
            Faces = new Face[facesCount];
            Name = name;
        }
    }
}
```



```

///<reference path="babylon.math.ts"/>

module SoftEngine {
    export interface Face {
        A: number;
        B: number;
        C: number;
    }

    export class Mesh {
        Position: BABYLON.Vector3;
        Rotation: BABYLON.Vector3;
        Vertices: BABYLON.Vector3[];
        Faces: Face[];

        constructor(public name: string, verticesCount: number, facesCount: number) {
            this.Vertices = new Array(verticesCount);
            this.Faces = new Array(facesCount);
            this.Rotation = new BABYLON.Vector3(0, 0, 0);
            this.Position = new BABYLON.Vector3(0, 0, 0);
        }
    }
}

```

```

var SoftEngine;
(function (SoftEngine) {
    var Mesh = (function () {
        function Mesh(name, verticesCount, facesCount) {
            this.name = name;
            this.Vertices = new Array(verticesCount);
            this.Faces = new Array(facesCount);
            this.Rotation = new BABYLON.Vector3(0, 0, 0);
            this.Position = new BABYLON.Vector3(0, 0, 0);
        }
        return Mesh;
    })();
    SoftEngine.Mesh = Mesh;
})(SoftEngine || (SoftEngine = {}));

```

We now need to update our **Render()** function/method of our **Device** object to iterate through all the faces defined and to draw the triangles associated.

- C#
- TypeScript

- JavaScript

```
foreach (var face in mesh.Faces)
{
    var vertexA = mesh.Vertices[face.A];
    var vertexB = mesh.Vertices[face.B];
    var vertexC = mesh.Vertices[face.C];

    var pixelA = Project(vertexA, transformMatrix);
    var pixelB = Project(vertexB, transformMatrix);
    var pixelC = Project(vertexC, transformMatrix);

    DrawLine(pixelA, pixelB);
    DrawLine(pixelB, pixelC);
    DrawLine(pixelC, pixelA);
}
```

```
for (var indexFaces = 0; indexFaces < cMesh.Faces.length; indexFaces++)
{
    var currentFace = cMesh.Faces[indexFaces];
    var vertexA = cMesh.Vertices[currentFace.A];
    var vertexB = cMesh.Vertices[currentFace.B];
    var vertexC = cMesh.Vertices[currentFace.C];

    var pixelA = this.project(vertexA, transformMatrix);
    var pixelB = this.project(vertexB, transformMatrix);
    var pixelC = this.project(vertexC, transformMatrix);

    this.drawLine(pixelA, pixelB);
    this.drawLine(pixelB, pixelC);
    this.drawLine(pixelC, pixelA);
}
```

```
for (var indexFaces = 0; indexFaces < cMesh.Faces.length; indexFaces++)
{
    var currentFace = cMesh.Faces[indexFaces];
    var vertexA = cMesh.Vertices[currentFace.A];
    var vertexB = cMesh.Vertices[currentFace.B];
    var vertexC = cMesh.Vertices[currentFace.C];

    var pixelA = this.project(vertexA, transformMatrix);
    var pixelB = this.project(vertexB, transformMatrix);
    var pixelC = this.project(vertexC, transformMatrix);

    this.drawLine(pixelA, pixelB);
    this.drawLine(pixelB, pixelC);
    this.drawLine(pixelC, pixelA);
}
```

We finally need to declare the mesh associated with our **Cube** properly with its **12 faces** to make this new code working as expected.

Here is the new declaration:

- C#
- TypeScript
- JavaScript

```

var mesh = new SoftEngine.Mesh("Cube", 8, 12);
meshes.Add(mesh);
mesh.Vertices[0] = new Vector3(-1, 1, 1);
mesh.Vertices[1] = new Vector3(1, 1, 1);
mesh.Vertices[2] = new Vector3(-1, -1, 1);
mesh.Vertices[3] = new Vector3(1, -1, 1);
mesh.Vertices[4] = new Vector3(-1, 1, -1);
mesh.Vertices[5] = new Vector3(1, 1, -1);
mesh.Vertices[6] = new Vector3(1, -1, -1);
mesh.Vertices[7] = new Vector3(-1, -1, -1);

mesh.Faces[0] = new Face { A = 0, B = 1, C = 2 };
mesh.Faces[1] = new Face { A = 1, B = 2, C = 3 };
mesh.Faces[2] = new Face { A = 1, B = 3, C = 6 };
mesh.Faces[3] = new Face { A = 1, B = 5, C = 6 };
mesh.Faces[4] = new Face { A = 0, B = 1, C = 4 };
mesh.Faces[5] = new Face { A = 1, B = 4, C = 5 };

mesh.Faces[6] = new Face { A = 2, B = 3, C = 7 };
mesh.Faces[7] = new Face { A = 3, B = 6, C = 7 };
mesh.Faces[8] = new Face { A = 0, B = 2, C = 7 };
mesh.Faces[9] = new Face { A = 0, B = 4, C = 7 };
mesh.Faces[10] = new Face { A = 4, B = 5, C = 6 };
mesh.Faces[11] = new Face { A = 4, B = 6, C = 7 };

```

```

var mesh = new SoftEngine.Mesh("Cube", 8, 12);
meshes.push(mesh);
mesh.Vertices[0] = new BABYLON.Vector3(-1, 1, 1);
mesh.Vertices[1] = new BABYLON.Vector3(1, 1, 1);
mesh.Vertices[2] = new BABYLON.Vector3(-1, -1, 1);
mesh.Vertices[3] = new BABYLON.Vector3(1, -1, 1);
mesh.Vertices[4] = new BABYLON.Vector3(-1, 1, -1);
mesh.Vertices[5] = new BABYLON.Vector3(1, 1, -1);
mesh.Vertices[6] = new BABYLON.Vector3(1, -1, -1);
mesh.Vertices[7] = new BABYLON.Vector3(-1, -1, -1);

mesh.Faces[0] = { A:0, B:1, C:2 };
mesh.Faces[1] = { A:1, B:2, C:3 };
mesh.Faces[2] = { A:1, B:3, C:6 };
mesh.Faces[3] = { A:1, B:5, C:6 };
mesh.Faces[4] = { A:0, B:1, C:4 };
mesh.Faces[5] = { A:1, B:4, C:5 };

mesh.Faces[6] = { A:2, B:3, C:7 };
mesh.Faces[7] = { A:3, B:6, C:7 };
mesh.Faces[8] = { A:0, B:2, C:7 };
mesh.Faces[9] = { A:0, B:4, C:7 };
mesh.Faces[10] = { A:4, B:5, C:6 };
mesh.Faces[11] = { A:4, B:6, C:7 };

```

```

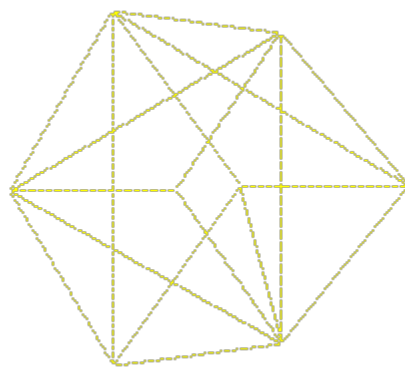
var mesh = new SoftEngine.Mesh("Cube", 8, 12);
meshes.push(mesh);
mesh.Vertices[0] = new BABYLON.Vector3(-1, 1, 1);
mesh.Vertices[1] = new BABYLON.Vector3(1, 1, 1);
mesh.Vertices[2] = new BABYLON.Vector3(-1, -1, 1);
mesh.Vertices[3] = new BABYLON.Vector3(1, -1, 1);
mesh.Vertices[4] = new BABYLON.Vector3(-1, 1, -1);
mesh.Vertices[5] = new BABYLON.Vector3(1, 1, -1);
mesh.Vertices[6] = new BABYLON.Vector3(1, -1, -1);
mesh.Vertices[7] = new BABYLON.Vector3(-1, -1, -1);

mesh.Faces[0] = { A:0, B:1, C:2 };
mesh.Faces[1] = { A:1, B:2, C:3 };
mesh.Faces[2] = { A:1, B:3, C:6 };
mesh.Faces[3] = { A:1, B:5, C:6 };
mesh.Faces[4] = { A:0, B:1, C:4 };
mesh.Faces[5] = { A:1, B:4, C:5 };

mesh.Faces[6] = { A:2, B:3, C:7 };
mesh.Faces[7] = { A:3, B:6, C:7 };
mesh.Faces[8] = { A:0, B:2, C:7 };
mesh.Faces[9] = { A:0, B:4, C:7 };
mesh.Faces[10] = { A:4, B:5, C:6 };
mesh.Faces[11] = { A:4, B:6, C:7 };

```

You should now have this beautiful rotating cube:



Congrats! 😊

Enhancing the line drawing algorithm with Bresenham

There is an optimized way to draw our lines using the Bresenham's line algorithm (http://en.wikipedia.org/wiki/Bresenham's_line_algorithm). It's faster & sharper than our current simple recursive version. The story of this algorithm is fascinating. Please read the Wikipedia definition of this algorithm to discover how Bresenham build it and for which reasons.

Here are the versions of this algorithm in C#, TypeScript and JavaScript:

- C#
- TypeScript
- JavaScript

```
public void DrawBline(Vector2 point0, Vector2 point1)
{
    int x0 = (int)point0.X;
    int y0 = (int)point0.Y;
    int x1 = (int)point1.X;
    int y1 = (int)point1.Y;

    var dx = Math.Abs(x1 - x0);
    var dy = Math.Abs(y1 - y0);
    var sx = (x0 < x1) ? 1 : -1;
    var sy = (y0 < y1) ? 1 : -1;
    var err = dx - dy;

    while (true) {
        DrawPoint(new Vector2(x0, y0));

        if ((x0 == x1) && (y0 == y1)) break;
        var e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 < dx) { err += dx; y0 += sy; }
    }
}
```

```

public drawBline(point0: BABYLON.Vector2, point1: BABYLON.Vector2)
: void {
    var x0 = point0.x >> 0;
    var y0 = point0.y >> 0;
    var x1 = point1.x >> 0;
    var y1 = point1.y >> 0;
    var dx = Math.abs(x1 - x0);
    var dy = Math.abs(y1 - y0);
    var sx = (x0 < x1) ? 1 : -1;
    var sy = (y0 < y1) ? 1 : -1;
    var err = dx - dy;

    while (true) {
        this.drawPoint(new BABYLON.Vector2(x0, y0));

        if ((x0 == x1) && (y0 == y1)) break;
        var e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 < dx) { err += dx; y0 += sy; }
    }
}

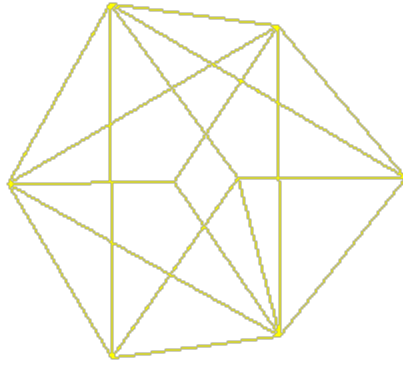
```

```

Device.prototype.drawBline = function (point0, point1) {
    var x0 = point0.x >> 0;
    var y0 = point0.y >> 0;
    var x1 = point1.x >> 0;
    var y1 = point1.y >> 0;
    var dx = Math.abs(x1 - x0);
    var dy = Math.abs(y1 - y0);
    var sx = (x0 < x1) ? 1 : -1;
    var sy = (y0 < y1) ? 1 : -1;
    var err = dx - dy;
    while(true) {
        this.drawPoint(new BABYLON.Vector2(x0, y0));
        if((x0 == x1) && (y0 == y1)) break;
        var e2 = 2 * err;
        if(e2 > -dy) { err -= dy; x0 += sx; }
        if(e2 < dx) { err += dx; y0 += sy; }
    }
};

```

In the render function, replace the call do DrawLine by DrawBline and you should notice that this is a bit more fluid & a bit more sharp:



If you're observing it with attention, you should see that this version using Bresenham is less choppy than our first algorithm.

Again, you can **download the solutions** containing the source code:

- **C#** : SoftEngineCSharpPart2.zip

(<http://david.blob.core.windows.net/softengine3d/SoftEngineCSharpPart2.zip>)

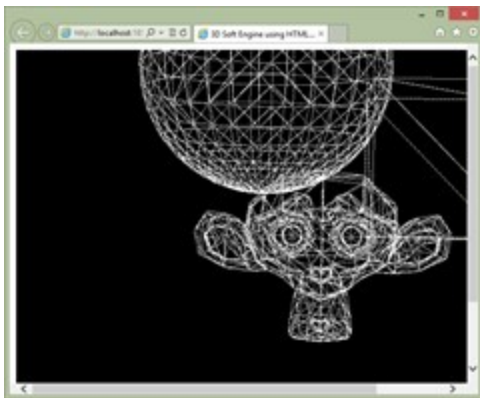
- **TypeScript** : SoftEngineTSPart2.zip

(<http://david.blob.core.windows.net/softengine3d/SoftEngineTSPart2.zip>)

- **JavaScript** : SoftEngineJSPart2.zip

(<http://david.blob.core.windows.net/softengine3d/SoftEngineJSPart2.zip>) or simply right-click -> view source on the embedded iframe

In next tutorial, you will learn how to **export some Meshes from Blender**, a free 3D modeler tool, into a JSON format. We will then **load this JSON file to display it with our wireframe engine**. Indeed, we already have everything setup to display much more complex meshes like these one:





(https://msdnshared.blob.core.windows.net/media/MSDNBlogsFS/prod.evol.blogs.msdn.com/CommunityServer.Blogs.Components.WeblogFiles/00/00/01/10/46/metablogapi/3323.image_08997040.png)


See you in the third part: learning how to write a 3D soft engine in C#, TS or JS – loading meshes exported from Blender
(<https://www.davrous.com/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/>)

Follow @davrous 4,499 followers

Share this:

 (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?share=twitter&nb=1>)

 (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?share=facebook&nb=1>)

 (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?share=google-plus-1&nb=1>)

Related

Tutorial series:
learning how to write
a 3D soft engine from
scratch in C#,
TypeScript or
JavaScript
(<https://www.davrous.com/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/>)

Tutorial part 3:
learning how to write
a 3D soft engine in C#,
TS or JS – loading
meshes exported
from Blender
(<https://www.davrous.com/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/>)

Tutorial part 4:
learning how to write
a 3D software engine
in C#, TS or JS –
Rasterization &
Z-Buffering
(<https://www.davrous.com/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-z-buffering/>)
June 21, 2013

June 13, 2013
In "3D Software
Engine"

June 17, 2013
In "3D Software
Engine"

In "3D Software
Engine"

Tagged 3DEngine (<https://www.davrou.com/tag/3dengine/>), C# (<https://www.davrou.com/tag/c/>), Canvas (<https://www.davrou.com/tag/canvas/>), HTML5 (<https://www.davrou.com/tag/html5/>), JavaScript (<https://www.davrou.com/tag/javascript/>), Tutorial (<https://www.davrou.com/tag/tutorial/>), TypeScript (<https://www.davrou.com/tag/typescript/>), Windows 8 (<https://www.davrou.com/tag/windows-8/>)

Tutorial series: learning how to write a 3D soft engine from scratch in C#, TypeScript or JavaScript (<https://www.davrou.com/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/>)

Tutorial part 3: learning how to write a 3D soft engine in C#, TS or JS – loading meshes exported from Blender (<https://www.davrou.com/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/>)

16 thoughts on “Tutorial part 2: learning how to write a 3D soft engine from scratch in C#, TS or JS – drawing lines & triangles”



dan.persson@outlook.com

logs.msdn.com/dan.persson_4000_outlook.com/ProfileUrlRedirect.as

June 15, 2013 at 11:53 am (<https://www.davrou.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-704>)

Awesome tutorial! Looking forward to part 3

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=704#respond>)



Ian says:

August 15, 2013 at 6:37 pm (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-703>)

These are amazing! Keep up the great work!

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=703#respond>)



WOW says:

January 3, 2014 at 11:00 pm (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-702>)

WOW

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=702#respond>)



Salva says:

January 28, 2014 at 11:21 am (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-701>)

Why do you prefer to draw triangles instead of quads? (I've searched for it on the internet but there's much controversy about it.)

Triangles are the most simple 2D shape with area and you can build whatever polygon from them, but you will have more triangles than quads.

Isn't a better solution to use arrays with a variable size which fit their size to the number of vertexes of each face?

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=701#respond>)



Salva says:

January 28, 2014 at 11:21 am (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-700>)

Why do you prefer to draw triangles instead of quads? (I've searched for it on the internet but there's much controversy about it.)

Triangles are the most simple 2D shape with area and you can build whatever polygon from them, but you will have more triangles than quads.

Isn't a better solution to use arrays with a variable size which fit their size to the number of vertexes of each face?

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=700#respond>)



JSnovice says:

June 14, 2016 at 1:16 pm (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-1509>)

Because triangles can only be on one plane, quads might mess up stuff, if not all 4 points are on 1 plane.

Quads are easier to implement, but if you want to do it clean, then use triangles.

Or in other words, if a quad isn't on a plane, the computer by itself doesn't know how to behave.

So the 3d model would look weird.

Yes I'm necroing this, but just to clear it up.

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=1509#respond>)



Liren Huang says:



February 12, 2014 at 8:44 am (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-699>)

David,

Thanks for your wonderful works! To calculate the middlePoint better to use

C#: `Vector2 middlePoint = (point0 + point1)/2;`

TypeScript: `var middlePoint = point0.add(point1).scale(0.5);`

Javar: `var middlePoint = point0.add(point1).scale(0.5);`

Looking forward to see more from you!

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=699#respond>)



Shalin says:

March 11, 2015 at 4:58 am (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/#comment-698>)

Very informative tutorial. Looking forward to part 3.

Regards,

Shalin

<http://creately.com> (<http://creately.com>)

Reply (<https://www.davrous.com/2013/06/14/tutorial-part-2-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-ts-or-js-drawing-lines-triangles/?replytocom=698#respond>)

Pingback: [Tutorial series: learning how to write a 3D soft engine from scratch in C#, TypeScript or JavaScript – David Rousset](https://davrous.com/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/)
(<https://davrous.com/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/>)

Pingback: [Tutorial series: learning how to write a 3D soft engine from scratch in C#, TypeScript or JavaScript | David Rousset – HTML5 & Gaming Technical Evangelist](https://blogs.msdn.microsoft.com/davrous/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/) (<https://blogs.msdn.microsoft.com/davrous/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript/>)

Pingback: [Tutorial part 3: learning how to write a 3D soft engine in C#, TS or JS – loading meshes exported from Blender | David Rousset – HTML5 & Gaming Technical Evangelist](https://blogs.msdn.microsoft.com/davrous/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/) (<https://blogs.msdn.microsoft.com/davrous/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/>)

Pingback: [Tutorial part 4: learning how to write a 3D software engine in C#, TS or JS – Rasterization & Z-Buffering | David Rousset – HTML5 & Gaming Technical Evangelist](https://blogs.msdn.microsoft.com/davrous/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-z-buffering/) (<https://blogs.msdn.microsoft.com/davrous/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-z-buffering/>)

Pingback: [Tutorial part 5: learning how to write a 3D software engine in C#, TS or JS – Flat & Gouraud Shading | David Rousset – HTML5 & Gaming Technical Evangelist](https://blogs.msdn.microsoft.com/davrous/2013/07/03/tutorial-part-5-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-flat-gouraud-shading/) (<https://blogs.msdn.microsoft.com/davrous/2013/07/03/tutorial-part-5-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-flat-gouraud-shading/>)

Pingback: [Tutorial part 3: learning how to write a 3D soft engine in C#, TS or JS – loading meshes exported from Blender – David Rousset](https://www.davrous.com/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/) (<https://www.davrous.com/2013/06/17/tutorial-part-3-learning-how-to-write-a-3d-soft-engine-in-c-ts-or-js-loading-meshes-exported-from-blender/>)

Pingback: [Tutorial part 6: learning how to write a 3D software engine in C#, TS or JS – Texture mapping, back-face culling & WebGL – David Rousset](https://www.davrous.com/2013/07/18/tutorial-part-6-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-texture-mapping-back-face-culling-webgl/)
(<https://www.davrous.com/2013/07/18/tutorial-part-6-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-texture-mapping-back-face-culling-webgl/>)

Pingback: [Tutorial part 4: learning how to write a 3D software engine in C#, TS or JS – Rasterization & Z-Buffering – David Rousset](https://www.davrous.com/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-z-buffering/)
(<https://www.davrous.com/2013/06/21/tutorial-part-4-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-rasterization-z-buffering/>)

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *


Email *

Website

Post Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.



Proudly powered by WordPress (<http://wordpress.org/>) | Theme: Oblique (<http://themeisle.com/themes/oblique/>) by Themeisle.