# 四，填充光栅化的三角形并使用深度缓冲

译者前言：

本文译自[MSDN](http://blogs.msdn.com/b/davrous/archive/2013/06/13/tutorial-series-learning-how-to-write-a-3d-soft-engine-from-scratch-in-c-typescript-or-javascript.aspx" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)，原作者为[David Rousset](https://social.msdn.microsoft.com/profile/david rousset/" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)，文章中如果有我的额外说明，我会加上【译者注：】。

正文开始：

在前面的教程中，我们学习了如何在C#、TypeScript或JavaScript中编写3D软件渲染引擎中的从Blender加载导出网格这一章节。

我们已经能够在引擎中加载从Blender导出的Json文件了。那么到现在为止，我们的渲染效果依然只是简单的线框渲染。但是，在本章我们将讲解如何使用三角形光栅化算法来填充三角形。然后，我们将使用深度缓冲，以避免在后面的面跑到前面来的问题。

本章教程是以下系列的一部分：

[1 – 编写相机、网格和设备对象的核心逻辑](http://blog.csdn.net/teajs/article/details/49989681" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

[2 – 绘制线段和三角形来获得线框渲染效果](http://blog.csdn.net/teajs/article/details/49998675" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

[3 – 加载通过Blender扩展导出JSON格式的网格](http://blog.csdn.net/teajs/article/details/50001659" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

4 –填充光栅化的三角形并使用深度缓冲（本文）

[4b – 额外章节：使用技巧和并行处理来提高性能](http://blog.csdn.net/teajs/article/details/50054509" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

[5 – 使用平面着色和高氏着色处理光](http://blogs.msdn.com/b/davrous/archive/2013/07/03/tutorial-part-5-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-flat-amp-gouraud-shading.aspx" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

[6 – 应用纹理、背面剔除以及一些WebGL相关](http://blogs.msdn.com/b/davrous/archive/2013/07/18/tutorial-part-6-learning-how-to-write-a-3d-software-engine-in-c-ts-or-js-texture-mapping-back-face-culling-amp-webgl.aspx" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

通过本章节，你将能够看到这样的效果：

[点击运行](http://david.blob.core.windows.net/softengine3d/part4/index.html" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

****光栅化****

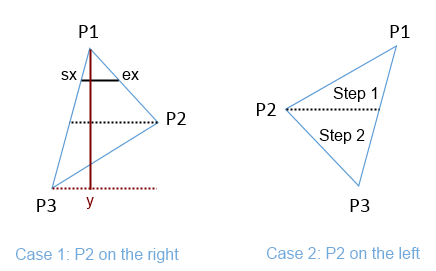
【译者注：感谢四川-平生小哥为我们翻译此段】

有很多不同类型的光栅化算法。我甚至知道在我的团队中有人向知名的GPU厂商提出了自己的光栅化算法。也多亏了他，我现在知道了什么是[折行书](https://en.wikipedia.org/wiki/Boustrophedon" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)并一直使用至今。 :-)

为了更正规，我们将在本教程中实现一个简单而有效的光栅化算法。正如我们在CPU中运行3D软件渲染引擎一般，它会耗费我们的大量CPU运算。当然，现今这个功能已经直接用GPU来帮我们完成。

先让我们做一个练习。请拿出一张纸，然后画一个三角形，嗯……任意你所能画出来的三角形。我们要找出一个通用的方法可以得出任意类型的三角形。

如果我们按Y轴对每个三角形的三个点进行排序，保证 P1 后面是 P2， 然后是 P3 的话，最终将出现两种可能的情况：



你将会看到这两种情况：

P2 在 P1 和 P3 的右侧 或 P2在 P1 和 P3 的左侧。在本教程中，由于我们始终是从左到右(sx 到 se)的顺序画线，所以就按照这个假设来处理这两种情况。

此外，我们要顺着左图中的红线自上而下 (从 P1.Y 到  P3.Y) 从左向右绘制。但是当到达P2.Y时我们需要稍微改变一下逻辑，因为这时两种情况的斜率都会发生改变。这就是为什么我们将扫描线处理分为两个步骤：从 P1.Y 向下移动到 P2.Y，然后从P2.Y 最终移动到 P3.Y。

要了解我们使用算法的全部逻辑，可以在维基百科中找到词条：[http://en.wikipedia.org/wiki/Slope](http://en.wikipedia.org/wiki/Slope" \o "http://en.wikipedia.org/wiki/Slope" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)。它只是一些基本的数学运算。

为了能够适应这两种情况，你只需要进行简单的运算：

dP1P2 = P2.X - P1.X / P2.Y - P1.Y

dP1P3 = P3.X - P1.X / P3.Y - P1.Y

那我们如何得知是属于哪种情况呢？

P2 在右的第一种情况：dP1P2 > dP1P3

P2 在左的第二种情况：dP1P3 > dP1P2

现在已经有了算法的基本逻辑，我们需要知道如何计算上图中每条线上的 sx(起始的 x 坐标) 和 ex(结束的 x 坐标) 之间的 x。因此要首先计算出 sx 和 ex。由于我们知道当前所扫描到的 y 值、P1P3 和 P1P3 的斜率，因此我们不难得出 sx 和 ex 值。

以情况1为例。首先利用当前的 y 值来计算梯度。它将告诉我们在 P1.Y 和 P2.Y之间进行处理时，我们当前所处的阶段。

梯度 = 当前的y值 - P1.Y / P2.Y - P1.Y

因为 x 和 y 是线性连接，所以我们可以基于该梯度，利用 P1.X 和 P3.X 来计算 sx 插值，并利用 P1.X 和 P2.X来计算 ex 插值。

如果您能够理解插值这个概念，那么你就能够理解剩下所有关于光纤和材质。而且你能够很好的阅读相关代码，也能够从头开始自己重写代码，而无须复制、粘贴下面的代码。

如果还不是很清楚的话，这里有一些关于光栅化的文章以供阅读：

[- 3D软件渲染引擎 - 第一部分](http://www.codeproject.com/Articles/170296/3D-Software-Rendering-Engine-Part-I" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

[- 三角形光栅化](https://lva.cg.tuwien.ac.at/ecg/wiki/doku.php?id=students:fill_rasterization" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

[- 填充三角形的软件光栅化算法](http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

现在，基于我们现有的算法描述说明让我们开始编写代码。首先，从设备对象删除 drawLine 和 drawBline 函数，并用下面的代码进行替换：

【译者注：C#代码】

// 将三维坐标和变换矩阵转换成二维坐标

public Vector3 Project(Vector3 coord, Matrix transMat){

// 进行坐标变换

var point = Vector3.TransformCoordinate(coord, transMat);

// 变换后的坐标起始点是坐标系的中心点

// 但是，在屏幕上，我们以左上角为起始点

// 我们需要重新计算使他们的起始点变成左上角

var x = point.X \* bmp.PixelWidth + bmp.PixelWidth / 2.0f;

var y = -point.Y \* bmp.PixelHeight + bmp.PixelHeight / 2.0f;

return (new Vector3(x, y, point.Z));}

// 如果二维坐标在可视范围内则绘制

public void DrawPoint(Vector2 point, Color4 color){

// 判断是否在屏幕内

if (point.X >= 0 && point.Y >= 0 && point.X < bmp.PixelWidth && point.Y < bmp.PixelHeight)

{

// 绘制一个点

PutPixel((int)point.X, (int)point.Y, color);

}}

【译者注：TypeScript代码】

// 将三维坐标和变换矩阵转换成二维坐标

public project(coord: BABYLON.Vector3, transMat: BABYLON.Matrix): BABYLON.Vector3 {

// 进行坐标变换

var point = BABYLON.Vector3.TransformCoordinates(coord, transMat);

// 变换后的坐标起始点是坐标系的中心点

// 但是，在屏幕上，我们以左上角为起始点

// 我们需要重新计算使他们的起始点变成左上角

var x = point.x \* this.workingWidth + this.workingWidth / 2.0;

var y = -point.y \* this.workingHeight + this.workingHeight / 2.0;

return (new BABYLON.Vector3(x, y, point.z));}

// 如果二维坐标在可视范围内则绘制

public drawPoint(point: BABYLON.Vector2, color: BABYLON.Color4): void {

// 判断是否在屏幕内

if(point.x >= 0 && point.y >= 0 && point.x < this.workingWidth && point.y < this.workingHeight) {

// 绘制一个点

this.putPixel(point.x, point.y, color);

}}

【译者注：JavaScript代码】

// 将三维坐标和变换矩阵转换成二维坐标

Device.prototype.project = function (coord, transMat) {

var point = BABYLON.Vector3.TransformCoordinates(coord, transMat);

// 变换后的坐标起始点是坐标系的中心点

// 但是，在屏幕上，我们以左上角为起始点

// 我们需要重新计算使他们的起始点变成左上角

var x = point.x \* this.workingWidth + this.workingWidth / 2.0 >> 0;

var y = -point.y \* this.workingHeight + this.workingHeight / 2.0 >> 0;

return (new BABYLON.Vector3(x, y, point.z));};

// 如果二维坐标在可视范围内则绘制

Device.prototype.drawPoint = function (point, color) {

// 判断是否在屏幕内

if (point.x >= 0 && point.y >= 0 && point.x < this.workingWidth

&& point.y < this.workingHeight) {

// 绘制一个点

this.putPixel(point.x, point.y, color);

}};

我们仅仅做了一些准备，下面则是最重要的部分，基于先前解释过的三角形的逻辑进行绘制。

【译者注：C#代码】

// 限制数值范围在0和1之间

float Clamp(float value, float min = 0, float max = 1){

return Math.Max(min, Math.Min(value, max));}

// 过渡插值

float Interpolate(float min, float max, float gradient){

return min + (max - min) \* Clamp(gradient);}

// 在两点之间从左到右绘制一条线段// papb -> pcpd// pa, pb, pc, pd在之前必须已经排好序

void ProcessScanLine(int y, Vector3 pa, Vector3 pb, Vector3 pc, Vector3 pd, Color4 color){

// 由当前的y值，我们可以计算出梯度

// 以此再计算出 起始X(sx) 和 结束X(ex)

// 如果pa.Y == pb.Y 或者 pc.Y== pd.y的话，梯度强制为1

var gradient1 = pa.Y != pb.Y ? (y - pa.Y) / (pb.Y - pa.Y) : 1;

var gradient2 = pc.Y != pd.Y ? (y - pc.Y) / (pd.Y - pc.Y) : 1;

int sx = (int)Interpolate(pa.X, pb.X, gradient1);

int ex = (int)Interpolate(pc.X, pd.X, gradient2);

// 从左(sx)向右(ex)绘制一条线

for (var x = sx; x < ex; x++)

{

DrawPoint(new Vector2(x, y), color);

}}

public void DrawTriangle(Vector3 p1, Vector3 p2, Vector3 p3, Color4 color){

// 进行排序，p1总在最上面，p2总在最中间，p3总在最下面

if (p1.Y > p2.Y)

{

var temp = p2;

p2 = p1;

p1 = temp;

}

if (p2.Y > p3.Y)

{

var temp = p2;

p2 = p3;

p3 = temp;

}

if (p1.Y > p2.Y)

{

var temp = p2;

p2 = p1;

p1 = temp;

}

// 反向斜率

float dP1P2, dP1P3;

// http://en.wikipedia.org/wiki/Slope

// 计算反向斜率

if (p2.Y - p1.Y > 0)

dP1P2 = (p2.X - p1.X) / (p2.Y - p1.Y);

else

dP1P2 = 0;

if (p3.Y - p1.Y > 0)

dP1P3 = (p3.X - p1.X) / (p3.Y - p1.Y);

else

dP1P3 = 0;

// 对于第一种情况来说，三角形是这样的：

// P1

// -

// --

// - -

// - -

// - - P2

// - -

// - -

// -

// P3

if (dP1P2 > dP1P3)

{

for (var y = (int)p1.Y; y <= (int)p3.Y; y++)

{

if (y < p2.Y)

{

ProcessScanLine(y, p1, p3, p1, p2, color);

}

else

{

ProcessScanLine(y, p1, p3, p2, p3, color);

}

}

}

// 对于第二种情况来说，三角形是这样的：

// P1

// -

// --

// - -

// - -

// P2 - -

// - -

// - -

// -

// P3

else

{

for (var y = (int)p1.Y; y <= (int)p3.Y; y++)

{

if (y < p2.Y)

{

ProcessScanLine(y, p1, p2, p1, p3, color);

}

else

{

ProcessScanLine(y, p2, p3, p1, p3, color);

}

}

}}

【译者注：TypeScript代码】

// 限制数值范围在0和1之间

public clamp(value: number, min: number = 0, max: number = 1): number {

return Math.max(min, Math.min(value, max));}

// 过渡插值

public interpolate(min: number, max: number, gradient: number) {

return min + (max - min) \* this.clamp(gradient);}

// 在两点之间从左到右绘制一条线段// papb -> pcpd// pa, pb, pc, pd在之前必须已经排好序

public processScanLine(y: number, pa: BABYLON.Vector3, pb: BABYLON.Vector3,

pc: BABYLON.Vector3, pd: BABYLON.Vector3, color: BABYLON.Color4): void {

// 由当前的y值，我们可以计算出梯度

// 以此再计算出 起始X(sx) 和 结束X(ex)

// 如果pa.Y == pb.Y 或者 pc.Y== pd.y的话，梯度强制为1

var gradient1 = pa.y != pb.y ? (y - pa.y) / (pb.y - pa.y) : 1;

var gradient2 = pc.y != pd.y ? (y - pc.y) / (pd.y - pc.y) : 1;

var sx = this.interpolate(pa.x, pb.x, gradient1) >> 0;

var ex = this.interpolate(pc.x, pd.x, gradient2) >> 0;

// 从左(sx)向右(ex)绘制一条线

for (var x = sx; x < ex; x++) {

this.drawPoint(new BABYLON.Vector2(x, y), color);

}}

public drawTriangle(p1: BABYLON.Vector3, p2: BABYLON.Vector3,

p3: BABYLON.Vector3, color: BABYLON.Color4): void {

// 进行排序，p1总在最上面，p2总在最中间，p3总在最下面

if (p1.y > p2.y) {

var temp = p2;

p2 = p1;

p1 = temp;

}

if (p2.y > p3.y) {

var temp = p2;

p2 = p3;

p3 = temp;

}

if (p1.y > p2.y) {

var temp = p2;

p2 = p1;

p1 = temp;

}

// 反向斜率

var dP1P2: number; var dP1P3: number;

// http://en.wikipedia.org/wiki/Slope

// 计算反向斜率

if (p2.y - p1.y > 0)

dP1P2 = (p2.x - p1.x) / (p2.y - p1.y);

else

dP1P2 = 0;

if (p3.y - p1.y > 0)

dP1P3 = (p3.x - p1.x) / (p3.y - p1.y);

else

dP1P3 = 0;

// 对于第一种情况来说，三角形是这样的：

// P1

// -

// --

// - -

// - -

// - - P2

// - -

// - -

// -

// P3

if (dP1P2 > dP1P3) {

for (var y = p1.y >> 0; y <= p3.y >> 0; y++)

{

if (y < p2.y) {

this.processScanLine(y, p1, p3, p1, p2, color);

}

else {

this.processScanLine(y, p1, p3, p2, p3, color);

}

}

}

// 对于第二种情况来说，三角形是这样的：

// P1

// -

// --

// - -

// - -

// P2 - -

// - -

// - -

// -

// P3

else {

for (var y = p1.y >> 0; y <= p3.y >> 0; y++)

{

if (y < p2.y) {

this.processScanLine(y, p1, p2, p1, p3, color);

}

else {

this.processScanLine(y, p2, p3, p1, p3, color);

}

}

}}

【译者注：JavaScript代码】

// 限制数值范围在0和1之间

Device.prototype.clamp = function (value, min, max) {

if (typeof min === "undefined") { min = 0; }

if (typeof max === "undefined") { max = 1; }

return Math.max(min, Math.min(value, max));};

// 过渡插值

Device.prototype.interpolate = function (min, max, gradient) {

return min + (max - min) \* this.clamp(gradient);};

// 在两点之间从左到右绘制一条线段// papb -> pcpd// pa, pb, pc, pd在之前必须已经排好序

Device.prototype.processScanLine = function (y, pa, pb, pc, pd, color) {

// 由当前的y值，我们可以计算出梯度

// 以此再计算出 起始X(sx) 和 结束X(ex)

// 如果pa.Y == pb.Y 或者 pc.Y== pd.y的话，梯度强制为1

var gradient1 = pa.y != pb.y ? (y - pa.y) / (pb.y - pa.y) : 1;

var gradient2 = pc.y != pd.y ? (y - pc.y) / (pd.y - pc.y) : 1;

var sx = this.interpolate(pa.x, pb.x, gradient1) >> 0;

var ex = this.interpolate(pc.x, pd.x, gradient2) >> 0;

// 从左(sx)向右(ex)绘制一条线

for (var x = sx; x < ex; x++) {

this.drawPoint(new BABYLON.Vector2(x, y), color);

}};

Device.prototype.drawTriangle = function (p1, p2, p3, color) {

// 进行排序，p1总在最上面，p2总在最中间，p3总在最下面

if (p1.y > p2.y) {

var temp = p2;

p2 = p1;

p1 = temp;

}

if (p2.y > p3.y) {

var temp = p2;

p2 = p3;

p3 = temp;

}

if (p1.y > p2.y) {

var temp = p2;

p2 = p1;

p1 = temp;

}

// 反向斜率

var dP1P2; var dP1P3;

// http://en.wikipedia.org/wiki/Slope

// 计算反向斜率

if (p2.y - p1.y > 0) {

dP1P2 = (p2.x - p1.x) / (p2.y - p1.y);

} else {

dP1P2 = 0;

}

if (p3.y - p1.y > 0) {

dP1P3 = (p3.x - p1.x) / (p3.y - p1.y);

} else {

dP1P3 = 0;

}

// 对于第一种情况来说，三角形是这样的：

// P1

// -

// --

// - -

// - -

// - - P2

// - -

// - -

// -

// P3

if (dP1P2 > dP1P3) {

for (var y = p1.y >> 0; y <= p3.y >> 0; y++) {

if (y < p2.y) {

this.processScanLine(y, p1, p3, p1, p2, color);

} else {

this.processScanLine(y, p1, p3, p2, p3, color);

}

}

}

// 对于第二种情况来说，三角形是这样的：

// P1

// -

// --

// - -

// - -

// P2 - -

// - -

// - -

// -

// P3

else {

for (var y = p1.y >> 0; y <= p3.y >> 0; y++) {

if (y < p2.y) {

this.processScanLine(y, p1, p2, p1, p3, color);

} else {

this.processScanLine(y, p2, p3, p1, p3, color);

}

}

}};

你已经了解了如何处理两种三角形的填写以及扫描线中所做的操作了。

最后，你需要更新渲染函数，用drawTriangle来替代drawLine和drawBline。我们还用了不同的灰色填充每个三角形。不然的话，整个画面一片灰你根本就看不出效果来。我们将在接下来的教程中学习到如何恰当的处理光照。

【译者注：C#代码】

var faceIndex = 0;

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);

var color = 0.25f + (faceIndex % mesh.Faces.Length) \* 0.75f / mesh.Faces.Length;

DrawTriangle(pixelA, pixelB, pixelC, new Color4(color, color, color, 1));

faceIndex++;}

【译者注：TypeScript代码】

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);

var color: number = 0.25 + ((indexFaces % cMesh.Faces.length) / cMesh.Faces.length) \* 0.75;

this.drawTriangle(pixelA, pixelB, pixelC, new BABYLON.Color4(color, color, color, 1));}

【译者注：JavaScript代码】

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);

var color = 0.25 + ((indexFaces % cMesh.Faces.length) / cMesh.Faces.length) \* 0.75;

this.drawTriangle(pixelA, pixelB, pixelC, new BABYLON.Color4(color, color, color, 1));}

结果应该是这样的：

[运行代码](http://david.blob.core.windows.net/softengine3d/part4sample1/index.html" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

这是怎么回事？为什么感觉这么奇怪？！嗯~这是因为我们没有正确的把正面的三角形画在正面。【译者注：我是这么翻译的，你就这么一看~】

如何使用深度缓冲

我们需要对当前的Z值在缓冲区中进行比较。

如果当前要绘制的像素Z值是最前面的（最靠近屏幕），则可以绘制。

然而，如果当前Z值大于前面的像素，则可以被丢弃。

我们需要一个东西用来保存深度缓冲区。因此，我们声明一个新的数组，并将其命名为深度缓冲区(depthBuffer)。该数组的大小等于 屏幕(width \* height)。

每次调用 clear() 函数时深度缓冲区内的每一个元素都需要一个非常高的默认Z值。

在putPixel(函数/方法)中，我们需要测试已存储在缓冲区中某个指定的像素Z值。此外，我们以前的逻辑接收Vector2用于绘制在屏幕上。现在我们将其改为Vector3用于增加Z值。因为现在我们将需要这部分新信息用于正确绘制面片。

最后，在我们的三角形内，同样需要一个类似 x 值插值的方式对 z 值进行插值。

总之，这里是你所需要对设备对象更新的代码：

【译者注：C#代码】

private byte[] backBuffer;

private readonly float[] depthBuffer;

private WriteableBitmap bmp;

private readonly int renderWidth;

private readonly int renderHeight;

public Device(WriteableBitmap bmp){

this.bmp = bmp;

renderWidth = bmp.PixelWidth;

renderHeight = bmp.PixelHeight;

// 后台缓冲区大小值是要绘制的像素

// 屏幕(width\*height) \* 4 (R,G,B & Alpha值)

backBuffer = new byte[bmp.PixelWidth \* bmp.PixelHeight \* 4];

depthBuffer = new float[bmp.PixelWidth \* bmp.PixelHeight];}

// 清除后台缓冲区为指定颜色

public void Clear(byte r, byte g, byte b, byte a){

// 清除后台缓冲区

for (var index = 0; index < backBuffer.Length; index += 4)

{

// Windows使用BGRA，而不是Html5中使用的RGBA

backBuffer[index] = b;

backBuffer[index + 1] = g;

backBuffer[index + 2] = r;

backBuffer[index + 3] = a;

}

// 清除深度缓冲区

for (var index = 0; index < depthBuffer.Length; index++)

{

depthBuffer[index] = float.MaxValue;

}}

// 调用此方法把一个像素绘制到指定的X, Y坐标上

public void PutPixel(int x, int y, float z, Color4 color){

// 我们的后台缓冲区是一维数组

// 这里我们简单计算，将X和Y对应到此一维数组中

var index = (x + y \* renderWidth);

var index4 = index \* 4;

if (depthBuffer[index] < z)

{

return; // 深度测试不通过

}

depthBuffer[index] = z;

backBuffer[index4] = (byte)(color.Blue \* 255);

backBuffer[index4 + 1] = (byte)(color.Green \* 255);

backBuffer[index4 + 2] = (byte)(color.Red \* 255);

backBuffer[index4 + 3] = (byte)(color.Alpha \* 255);}

// 将三维坐标和变换矩阵转换成二维坐标

public Vector3 Project(Vector3 coord, Matrix transMat){

// 进行坐标变换

var point = Vector3.TransformCoordinate(coord, transMat);

// 变换后的坐标起始点是坐标系的中心点

// 但是，在屏幕上，我们以左上角为起始点

// 我们需要重新计算使他们的起始点变成左上角

var x = point.X \* bmp.PixelWidth + bmp.PixelWidth / 2.0f;

var y = -point.Y \* bmp.PixelHeight + bmp.PixelHeight / 2.0f;

return (new Vector3(x, y, point.Z));}

// 如果二维坐标在可视范围内则绘制

public void DrawPoint(Vector3 point, Color4 color){

// 判断是否在屏幕内

if (point.X >= 0 && point.Y >= 0 && point.X < bmp.PixelWidth && point.Y < bmp.PixelHeight)

{

// 绘制一个点

PutPixel((int)point.X, (int)point.Y, point.Z, color);

}}

// 在两点之间从左到右绘制一条线段// papb -> pcpd// pa, pb, pc, pd在之前必须已经排好序

void ProcessScanLine(int y, Vector3 pa, Vector3 pb, Vector3 pc, Vector3 pd, Color4 color){

// 由当前的y值，我们可以计算出梯度

// 以此再计算出 起始X(sx) 和 结束X(ex)

// 如果pa.Y == pb.Y 或者 pc.Y== pd.y的话，梯度强制为1

var gradient1 = pa.Y != pb.Y ? (y - pa.Y) / (pb.Y - pa.Y) : 1;

var gradient2 = pc.Y != pd.Y ? (y - pc.Y) / (pd.Y - pc.Y) : 1;

int sx = (int)Interpolate(pa.X, pb.X, gradient1);

int ex = (int)Interpolate(pc.X, pd.X, gradient2);

// 计算 开始Z值 和 结束Z值

float z1 = Interpolate(pa.Z, pb.Z, gradient1);

float z2 = Interpolate(pc.Z, pd.Z, gradient2);

// 从左(sx)向右(ex)绘制一条线

for (var x = sx; x < ex; x++)

{

float gradient = (x - sx) / (float)(ex - sx);

var z = Interpolate(z1, z2, gradient);

DrawPoint(new Vector3(x, y, z), color);

}}

【译者注：TypeScript代码】

// 后台缓冲区大小值是要绘制的像素 // 屏幕(width\*height) \* 4 (R,G,B & Alpha值)

private backbuffer: ImageData;

private workingCanvas: HTMLCanvasElement;

private workingContext: CanvasRenderingContext2D;

private workingWidth: number;

private workingHeight: number;// 等同于backbuffer.data

private backbufferdata;

private depthbuffer: number[];

constructor(canvas: HTMLCanvasElement) {

this.workingCanvas = canvas;

this.workingWidth = canvas.width;

this.workingHeight = canvas.height;

this.workingContext = this.workingCanvas.getContext("2d");

this.depthbuffer = new Array(this.workingWidth \* this.workingHeight);}

// 用指定颜色清除后台缓冲区

public clear(): void {

// 使用默认颜色清除后台缓冲区

this.workingContext.clearRect(0, 0, this.workingWidth, this.workingHeight);

// 缓存后台缓冲区

this.backbuffer = this.workingContext.getImageData(0, 0, this.workingWidth, this.workingHeight);

// 清除深度缓冲区

for (var i = 0; i < this.depthbuffer.length; i++) {

// 使用最大值填充

this.depthbuffer[i] = 10000000;

}}

// 调用此方法把一个像素绘制到指定的X, Y坐标上

public putPixel(x: number, y: number, z: number, color: BABYLON.Color4): void {

this.backbufferdata = this.backbuffer.data;

// 我们的后台缓冲区是一维数组

// 这里我们简单计算，将X和Y对应到此一维数组中

var index: number = ((x >> 0) + (y >> 0) \* this.workingWidth);

var index4: number = index \* 4;

if (this.depthbuffer[index] < z) {

return; // 深度测试不通过

}

this.depthbuffer[index] = z;

// 在Html5 canvas中使用RGBA颜色空间

this.backbufferdata[index4] = color.r \* 255;

this.backbufferdata[index4 + 1] = color.g \* 255;

this.backbufferdata[index4 + 2] = color.b \* 255;

this.backbufferdata[index4 + 3] = color.a \* 255;}

// 将三维坐标和变换矩阵转换成二维坐标

public project(coord: BABYLON.Vector3, transMat: BABYLON.Matrix): BABYLON.Vector3 {

// 进行坐标变换

var point = BABYLON.Vector3.TransformCoordinates(coord, transMat);

// 变换后的坐标起始点是坐标系的中心点

// 但是，在屏幕上，我们以左上角为起始点

// 我们需要重新计算使他们的起始点变成左上角

var x = point.x \* this.workingWidth + this.workingWidth / 2.0;

var y = -point.y \* this.workingHeight + this.workingHeight / 2.0;

return (new BABYLON.Vector3(x, y, point.z));}

// 如果二维坐标在可视范围内则绘制

public drawPoint(point: BABYLON.Vector3, color: BABYLON.Color4): void {

// 判断是否在屏幕内

if (point.x >= 0 && point.y >= 0 && point.x < this.workingWidth && point.y < this.workingHeight) {

// 绘制一个点

this.putPixel(point.x, point.y, point.z, color);

}}

// 在两点之间从左到右绘制一条线段// papb -> pcpd// pa, pb, pc, pd在之前必须已经排好序

public processScanLine(y: number, pa: BABYLON.Vector3, pb: BABYLON.Vector3, pc: BABYLON.Vector3, pd: BABYLON.Vector3, color: BABYLON.Color4): void {

// 由当前的y值，我们可以计算出梯度

// 以此再计算出 起始X(sx) 和 结束X(ex)

// 如果pa.Y == pb.Y 或者 pc.Y== pd.y的话，梯度强制为1

var gradient1 = pa.y != pb.y ? (y - pa.y) / (pb.y - pa.y) : 1;

var gradient2 = pc.y != pd.y ? (y - pc.y) / (pd.y - pc.y) : 1;

var sx = this.interpolate(pa.x, pb.x, gradient1) >> 0;

var ex = this.interpolate(pc.x, pd.x, gradient2) >> 0;

// 计算 开始Z值 和 结束Z值

var z1: number = this.interpolate(pa.z, pb.z, gradient1);

var z2: number = this.interpolate(pc.z, pd.z, gradient2);

// 从左(sx)向右(ex)绘制一条线

for (var x = sx; x < ex; x++) {

var gradient: number = (x - sx) / (ex - sx); // 规范从左往右绘制

var z = this.interpolate(z1, z2, gradient);

this.drawPoint(new BABYLON.Vector3(x, y, z), color);

}}

【译者注：JavaScript代码】

function Device(canvas) {

this.workingCanvas = canvas;

this.workingWidth = canvas.width;

this.workingHeight = canvas.height;

this.workingContext = this.workingCanvas.getContext("2d");

this.depthbuffer = new Array(this.workingWidth \* this.workingHeight);}

// 用指定颜色清除后台缓冲区

Device.prototype.clear = function () {

// 使用默认颜色清除后台缓冲区

this.workingContext.clearRect(0, 0, this.workingWidth, this.workingHeight);

// 缓存后台缓冲区

this.backbuffer = this.workingContext.getImageData(0, 0, this.workingWidth, this.workingHeight);

// 清除深度缓冲区

for (var i = 0; i < this.depthbuffer.length; i++) {

// 使用最大值填充

this.depthbuffer[i] = 10000000;

}};

// 调用此方法把一个像素绘制到指定的X, Y坐标上

Device.prototype.putPixel = function (x, y, z, color) {

this.backbufferdata = this.backbuffer.data;

// 我们的后台缓冲区是一维数组

// 这里我们简单计算，将X和Y对应到此一维数组中

var index = ((x >> 0) + (y >> 0) \* this.workingWidth);

var index4 = index \* 4;

if (this.depthbuffer[index] < z) {

return; // 深度测试不通过

}

this.depthbuffer[index] = z;

// 在Html5 canvas中使用RGBA颜色空间

this.backbufferdata[index4] = color.r \* 255;

this.backbufferdata[index4 + 1] = color.g \* 255;

this.backbufferdata[index4 + 2] = color.b \* 255;

this.backbufferdata[index4 + 3] = color.a \* 255;};

// 将三维坐标和变换矩阵转换成二维坐标

Device.prototype.project = function (coord, transMat) {

// 进行坐标变换

var point = BABYLON.Vector3.TransformCoordinates(coord, transMat);

// 变换后的坐标起始点是坐标系的中心点

// 但是，在屏幕上，我们以左上角为起始点

// 我们需要重新计算使他们的起始点变成左上角

var x = point.x \* this.workingWidth + this.workingWidth / 2.0;

var y = -point.y \* this.workingHeight + this.workingHeight / 2.0;

return (new BABYLON.Vector3(x, y, point.z));};

// 如果二维坐标在可视范围内则绘制

Device.prototype.drawPoint = function (point, color) {

// 判断是否在屏幕内

if (point.x >= 0 && point.y >= 0 && point.x < this.workingWidth && point.y < this.workingHeight) {

// 绘制一个点

this.putPixel(point.x, point.y, point.z, color);

}};

// 在两点之间从左到右绘制一条线段// papb -> pcpd// pa, pb, pc, pd在之前必须已经排好序

Device.prototype.processScanLine = function (y, pa, pb, pc, pd, color) {

// 由当前的y值，我们可以计算出梯度

// 以此再计算出 起始X(sx) 和 结束X(ex)

// 如果pa.Y == pb.Y 或者 pc.Y== pd.y的话，梯度强制为1

var gradient1 = pa.y != pb.y ? (y - pa.y) / (pb.y - pa.y) : 1;

var gradient2 = pc.y != pd.y ? (y - pc.y) / (pd.y - pc.y) : 1;

var sx = this.interpolate(pa.x, pb.x, gradient1) >> 0;

var ex = this.interpolate(pc.x, pd.x, gradient2) >> 0;

// 计算 开始Z值 和 结束Z值

var z1 = this.interpolate(pa.z, pb.z, gradient1);

var z2 = this.interpolate(pc.z, pd.z, gradient2);

// 从左(sx)向右(ex)绘制一条线

for (var x = sx; x < ex; x++) {

var gradient = (x - sx) / (ex - sx);

var z = this.interpolate(z1, z2, gradient);

this.drawPoint(new BABYLON.Vector3(x, y, z), color);

}};

使用这些新代码，你将获得的效果：

[运行代码](http://david.blob.core.windows.net/softengine3d/part4/index.html" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

同样的，你可以下载源代码：

C#：[SoftEngineCSharpPart4.zip](http://david.blob.core.windows.net/softengine3d/SoftEngineCSharpPart4.zip" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

TypeScript：[SoftEngineTSPart4.zip](http://david.blob.core.windows.net/softengine3d/SoftEngineTSPart4.zip" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank)

JavaScript：[SoftEngineJSPart4.zip](http://david.blob.core.windows.net/softengine3d/SoftEngineJSPart4.zip" \t "https://www.kancloud.cn/digest/soft-3d-engine/_blank) 或只需右键点击 -> 查看框架的源代码

在下一章节，第五个教程中，我们将看到如何模拟光照着色效果，我们将得到这样的画面：

