ویرایش ۱۳۹۲

علی خاقانی اصل

khaghaniasl.ir

گرافیک کامپیوتری

با رویکرد برنامه نویسی مبتنی بر DirectX

فهرست مطالب

فصل اول: مبانی ریاضی گرافیک

فصل دوم: برنامه نویسی DirectX

فصل سوم: ساخت محیطهای سه بعدی

فصل اول: مبانی ریاضی گرافیک

مقدمه

هنگام رسم انواع تصاویر گرافیکی به خصوص انیمیشن های کامپیوتری، ممکن است لازم باشد اعمال مختلفی از قبیل تغییر زاویه دید، تغییر اندازه اشکال، تغییر مکان اشکال موجود در صحنه و... صورت گیرد. به عنوان مثال در یک نرم افزار پردازش تصاویر یکی از ضروری ترین کارها، بزرگنمائی تصویر جهت مشاهده جزئیات آن است یا در یک انیمیشن کامپیوتری، تغییر محل دوربین یا اشیا موجود در صحنه غیرقابل اجتناب است. تغییر در اندازه، جهت و موقعیت اشکال رسم شده در صحنه به کمک تبدیلات هندسی انجام می گیرد. سه تبدیل پایه ای که با استفاده از آنها می توان هر نوع تبدیل دیگری را انجام داد؛ انتقال، دوران و تغییر مقیاس می باشد. دو تبدیل مهم دیگر که می-توان آنها را به کمک تبدیلات پایه ای ایجاد کرد، انعکاس و برش می باشد.

ماتریسها اساس کار در DirectX هستند. شاید در مورد دلیل استفاده ماتریسها در DirectX ایده هائی داشته باشید، با این حال در این فصل در مورد ماتریسها و کاربرد آنها در گرافیک دو بعدی و سه بعدی بحث خواهد شد. البته هنگام برنامه نویسی با DirectX نیازی به انجام محاسبات ریاضی بر روی ماتریسها توسط برنامه نویس وجود ندارد اما اطلاع از عملیاتهای انجام شده در پشت صحنه می تواند مفید باشد. در این فصل ابتدا در مورد سه عمل ریاضی ساده اما اصلی بر روی ماتریسها بحث می کنیم سپس در مورد ماتریسهای همگن (که عملا توسط DirectX استفاده می شوند) صحبت خواهد شد.

۱-۱ تبدیلات پایه ای[1]

سه تبدیل پایه ای که با استفاده از آنها می توان هر نوع تبدیل دیگری را انجام داد، انتقال، دوران و تغییر مقیاس می‌باشد. در ادامه هر یک از این تبدیلات در فضای دو بعدی را بررسی می کنیم:

۱-۱-۱ انتقال[2]

انجام عمل انتقال بر روی یک شی باعث تغییر محل آن در امتداد یک خط راست به محل دیگری می شود. جهت تغییر محل یک نقطه در فضای دو بعدی با مختصات (x,y) کافی است مسافت تغییر موقعیت در طول هر یک از محورهای x و y را به مختصات اولیه آن اضافه کنیم. این مسافتها را با t_x و t_y نشان می دهیم.

$$x' = x + t_x$$
$$y' = y + t_y$$

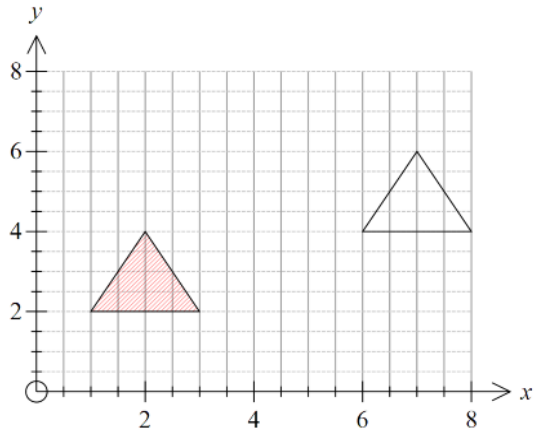
در صورتی که از نمایش ماتریسی برای نقاط استفاده کنیم خواهیم داشت:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

انتقال یک تبدیل یکنواخت است یعنی انتقال هر شکل به هر مختصات جدیدی تغییری در خود شکل ایجاد نمی-کند. انتقال یک خط راست می تواند با انتقال نقاط دو سر آن و سپس رسم خط ما بین نقاط جدید صورت گیرد. انتقال یک چند ضلعی نیز می تواند با انتقال هر یک از رئوس آن انجام شود.

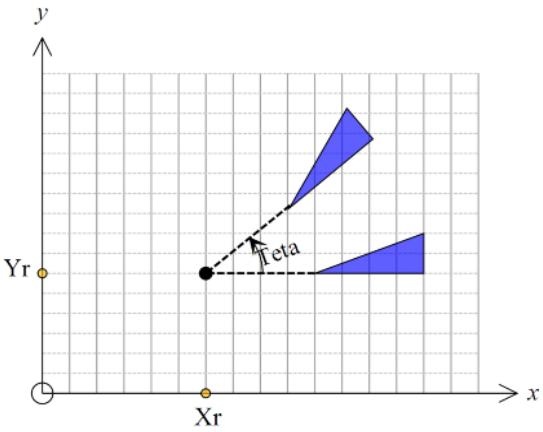
---

[1] Basic Transformation
[2] Translation



۱-۱-۲ دوران[۳]

دوران یک نقطه در فضای دو بعدی با تغییر مختصات آن در امتداد یک مسیر دایره ای انجام می شود. برای انجام عمل دوران زاویه دوران یا θ و نقطه ای که دوران حول آن انجام خواهد شد، باید معلوم باشد. مقادیر مثبت برای θ باعث دوران در جهت پادساعتگرد و مقادیر منفی برای θ باعث دوران در جهت ساعتگرد خواهد شد. دوران می تواند حول یکی از محورهای مختصات نیز انجام گیرد.



دوران یک نقطه حول مبدا مختصات با زاویه θ طبق رابطه زیر صورت می گیرد:

$$x' = x.\cos\theta - y.\sin\theta$$
$$y' = x.\sin\theta + y.\cos\theta$$

در صورتی که از نمایش ماتریسی برای نقاط استفاده کنیم ماتریس دوران به صورت خواهد $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ بود که با ضرب آن در مختصات اولیه نقطه مختصات جدید آن به دست خواهد آمد.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

دوران حول یک نقطه دلخواه مانند $[x_r, y_r]$ نیز با استفاده از رابطه کلی زیر انجام می شود:

$$x' = x_r + (x - x_r).\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r).\sin\theta + (y - y_r)\cos\theta$$

در رابطه فوق اگر به جای x_r و y_r مقدار صفر قرار دهیم همان رابطه دوران حول مبدا به دست خواهد آمد.

---

[3] Rotation

همانند انتقال، دوران نیز یک تبدیل یکنواخت بوده و ظاهر یک شی پس از دوران به همان شکل قبلی باقی می-ماند. لذا دوران خطوط و چند ضلعی ها می تواند با دوران رئوس آنها انجام گیرد.

۱-۱-۳ تغییر مقیاس[۴]

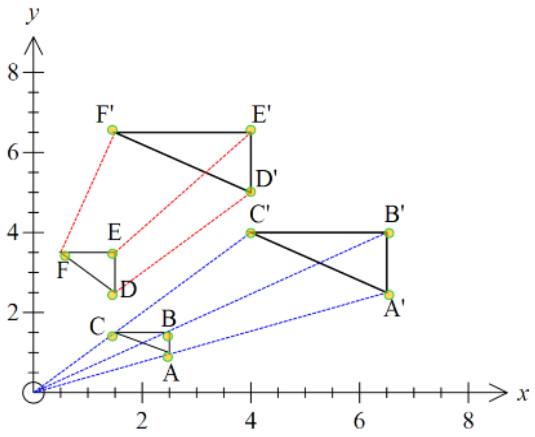
این تبدیل باعث تغییر در اندازه یک شی می شود. تغییر مقیاس چند ضلعی ها با ضرب مختصات رئوس آنها در ضرایب تغییر مقیاس s_x و s_y انجام می شود:

$$x' = x.s_x$$
$$y' = y.s_y$$

s_x باعث تغییر در عرض شکل (در راستای محور x ها) و s_y باعث تغییر در طول شکل (در راستای محور yها) می‌شود. در صورتی که از نمایش ماتریسی برای نقاط استفاده شود خواهیم داشت:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

ماتریس تغییر مقیاس در فضای دو بعدی است. هر عدد بزرگتر از صفری می تواند برای مقادیر s_x و s_y انتخاب شود. مقادیر بزرگتر از یک باعث افزایش اندازه شی و مقادیر بین صفر و یک باعث کوچکتر شدن اندازه شی خواهد شد. در صورتی که مقادیر s_x و s_y یکسان انتخاب شوند تغییر مقیاس یکنواخت خواهد بود یعنی ظاهر شی هنگام بزرگتر یا کوچکتر شدن تغییر نخواهد کرد. در صورتی که مقادیر s_x و s_y متفاوت باشند تغییراتی در شکل به وجود خواهد آمد. به عنوان مثال مربع به شکل مستطیل در می آید.



با استفاده از رابطه فوق هنگام تغییر مقیاس، موقعیت اشیا نیز تغییر خواهد کرد. اگر برای s_x و s_y مقدار کمتر از یک انتخاب شود، اشیا حین تغییر مقیاس به مرکز مختصات نزدیکتر شده و در صورتی که برای s_x و s_y مقدار بزرگتر از یک انتخاب شود هنگام تغییر مقیاس اشیا از مرکز مختصات دورتر می شوند. می توان نقطه ای مانند $[x_f, y_f]$ را انتخاب کرد که فاصله شکل پس از تغییر مقیاس نسبت به آن ثابت باقی بماند. این نقطه می تواند مختصات یکی از رئوس شکل و یا هر مختصات دلخواه دیگری باشد. در این صورت رابطه تغییر مقیاس به شکل زیر خواهد بود

$$x' = x.s_x + x_f(1 - s_x)$$
$$y' = y.s_y + y_f(1 - s_y)$$

در رابطه فوق مقدار $x_f(1 - s_x)$ و $y_f(1 - s_y)$ برای تمام نقاط یک شی ثابت خواهد بود.

---

[4] Scaling

۱-۲ ماتریسهای همگن دو بعدی

بسیاری از نرم افزارهای گرافیکی مجموعه ای از تبدیلات متوالی را بر روی اشیا موجود در صحنه اعمال می کنند. به عنوان مثال در یک انیمیشن اشیا مختلف به صورت پیوسته حرکت کرده (انتقال)، فاصله خود را با دوربین تغییر داده (تغییر مقیاس) و یا از زاویه دیگری مشاهده می‌شوند (دوران). حتی گاهی لازم است که دو یا هر سه نوع تبدیل به طور همزمان بر روی یک شی اعمال شود. با توجه به مطالب بخش قبل می توان از رابطه کلی زیر برای هر یک از تبدیلات پایه ای استفاده کرد:

$$p' = M_1.p + M_2$$

در این رابطه p مختصات اولیه یک نقطه یا نقاط یک شی و p' مختصات جدید آن است. ماتریس M_1 یک ماتریس۲*۲ برای تبدیلات دوران یا تغییر مقیاس و ماتریس M_2 ماتریس ۲*۲ دیگری برای انتقال است.

در این روش اعمال چندین تبدیل مختلف به صورت همزمان بر روی شی امکان پذیر نخواهد بود. مثلا برای انجام عمل تغییر مقیاس، دوران و انتقال بر روی یک شی ابتدا باید ماتریس M_2 در رابطه فوق را صفر در نظر گرفته و ماتریس تغییر مقیاس را در p ضرب نمود. سپس دوباره با صفر گرفتن ماتریس M_2، ماتریس دوران را در مختصات جدید به دست آمده ضرب کرده و در نهایت ماتریس انتقال M_2 را به مختصات جدید حاصل، اضافه نمود. بدیهی است اگر تعداد نقاط شی زیاد بوده و یا بخواهیم این عمل را بر روی تعداد زیادی شی انجام دهیم، محاسبات سنگینی بر سیستم تحمیل خواهد شد.

روش بسیار کاراتر، ترکیب تمام تبدیلات با یکدیگر و اعمال تبدیل نهائی بر روی شی به صورت یک تبدیل واحد می باشد. برای انجام این عمل جمله جمع با M_2از رابطه فوق باید حذف گردد. با تبدیل ماتریس M_1 به یک ماتریس ۳*۳ می توان ماتریس M_2 از رابطه فوق را حذف کرد. برای این منظور به جای نمایش مختصات نقاط به صورت (x,y) آنها را در قالب مختصات همگن به صورت (x_h, y_h, h) نشان می‌دهیم که در آن

$$x = \frac{x_h}{h}$$

$$y = \frac{y_h}{h}$$

می باشد. برای یک نقطه مفروض می توان هر مقدار غیرصفری را برای h اختیار کرد. در نتیجه برای هر نقطه می-توان بینهایت مختصات همگن تعریف نمود. معمولا h=1 انتخاب شده و در نتیجه مختصات همگن نقطه برابر (x,y,1) می شود اما در تبدیلات سه بعدی گاهی لازم است مقادیر دیگری برای h انتخاب شود.

با استفاده از مختصات همگن می توان تمام تبدیلات پایه ای را با ضرب ماتریس تبدیل در مختصات نقطه انجام داد. در مختصات همگن هر نقطه با استفاده از یک ماتریس ۳*۱ نشان داده شده و ماتریسهای تبدیل نیز همگی ۳*۳ خواهند بود.

برای انتقال خواهیم داشت:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

دوران حول مبدا با استفاده از مختصات همگن نیز به شکل زیر خواهد بود:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

تغییر مقیاس با استفاده از مختصات همگن نیز به صورت زیر قابل انجام خواهد بود:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

همانطور که ملاحظه می شود هر سه تبدیل پایه ای با ضرب مختصات همگن نقطه در یک ماتریس تبدیل ۳*۳ قابل انجام است.

۱-۳ تبدیلات مرکب

با استفاده از مختصات همگن می توان تبدیلات مرکب (انجام چند تبدیل همزمان بر روی شی) را بسیار کاراتر انجام داد. برای انجام چند تبدیل به صورت یکجا بر روی یک شی می توان ابتدا هر یک از ماتریسهای تبدیل را در یکدیگر ضرب و ماتریس تبدیل نهائی را در مختصات نقاط شی ضرب کرد.

از آنجایی که همه ماتریسهای تبدیل در فضای دو بعدی ۳*۳ می باشند، ضرب آنها در یکدیگر هزینه چندانی نخواهد داشت بنابراین انجام تبدیلات مرکب بسیار سریعتر صورت خواهد گرفت. به عنوان مثال اگر بخواهیم دو انتقال با ماتریسهای T_1 و T_2 را بر روی یک شی مانند p اعمال کنیم خواهیم داشت:

$$\begin{aligned} p' &= T_2 * (T_1 * p) \\ &= T_2 * T_1 * p \\ &= (T_2 * T_1) * p \end{aligned}$$

با ضرب دو ماتریس تبدیل T_1 و T_2 در یکدیگر خواهیم داشت:

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x2} + t_{x1} \\ 0 & 1 & t_{y2} + t_{y1} \\ 0 & 0 & 1 \end{bmatrix}$$

همانطور که ملاحظه می شود ضرب دو ماتریس انتقال در یکدیگر ماتریس انتقالی را نتیجه خواهد داد که هر دو انتقال را شامل می شود. به سادگی می توان درستی این رابطه را برای سایر تبدیلات نیز نشان داد. در حالت کلی برای ترکیب هر تعداد تبدیل با هر نوعی می توان ماتریسهای تبدیل را در یکدیگر ضرب کرد.

۱-۳-۱ دوران حول نقطه دلخواه

جهت دوران دادن یک شی حول هر نقطه دلخواه مانند (x_r, y_r) می توان از روش ساده زیر استفاده کرد:

۱. انتقال شی به مختصاتی که نقطه (x_r, y_r) نسبت به آن در مرکز مختصات قرار گیرد.
۲. دوران شی حول مبدا مختصات
۳. انتقال شی به مختصاتی که نقطه (x_r, y_r) در موقعیت اولیه اش قرار گیرد.

با توجه به مطالب فوق ماتریس دوران یک شی حول نقطه دلخواه (x_r, y_r) با زاویه θ به شکل زیر خواهد بود:

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

۱-۳-۲ تغییر مقیاس در مختصات ثابت

همان طور که قبلا اشاره شد تغییر مقیاس یک شی باعث تغییر موقعیت آن نیز می شود. اگر بخواهیم فاصله شی پس از تغییر اندازه نسبت به نقطه ای مانند (x_r, y_r) (مثلا نسبت به مرکز خودش) ثابت باقی بماند می توان از روش زیر استفاده کرد:

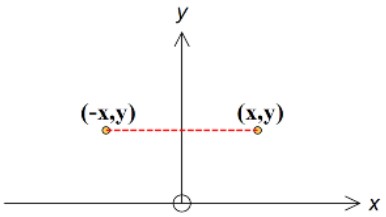
۱. انتقال شی به گونه ای که نقطه (x_r, y_r) نسبت به آن در مرکز مختصات قرار گیرد
۲. تغییر مقیاس شی
۳. انتقال شی به گونه ای که (x_r, y_r) در موقعیت اولیه اش قرار گیرد

بنابراین ماتریس تغییر مقیاس یک شی با حفظ فاصله آن از نقطه (x_r, y_r) به شکل زیر خواهد بود:

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

۱-۳-۳ انعکاس[5]

یکی از تبدیلات رایج که جزو تبدیلات اصلی محسوب نشده اما کاربرد زیادی دارد انعکاس نسبت به یکی از محورهای مختصات است. برای انجام عمل انعکاس بر روی یک نقطه نسبت به محور x کافی است مقدار مختصات y آن را به y- تغییر دهیم و برای انعکاس حول محور y کافی است مختصات x نقطه را به x- تغییر دهیم.



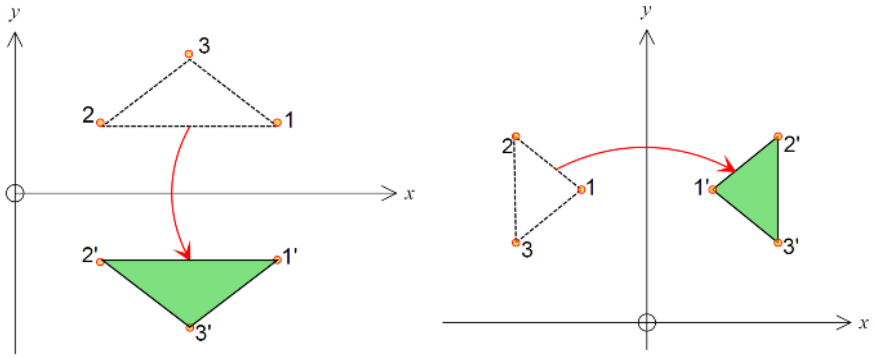
بنابراین برای انعکاس حول محور y خواهیم داشت: $x' = -x, y' = y$ و برای انعکاس حول محور x خواهیم داشت: $x' = x, y' = -y$. اگر از نمایش ماتریسی برای نقاط استفاده کنیم برای انعکاس روی محور y خواهیم داشت:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix}$$

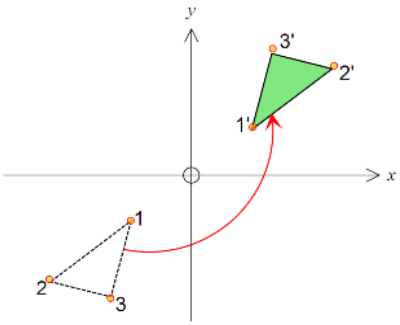
و برای انعکاس روی محور x نیز خواهیم داشت:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ -y \end{bmatrix}$$

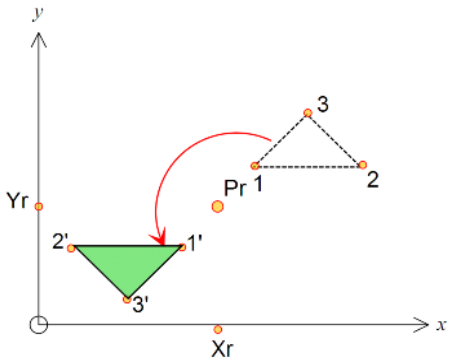
در مختصات همگن ماتریس انعکاس نسبت به محور x به صورت $\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ و نسبت به محور y به صورت $\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ و نسبت به مبدا مختصات به صورت $\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ خواهد بود. در شکلهای زیر انعکاس یک مثلث در هر سه حالت نشان داده شده است:



[5] Reflection

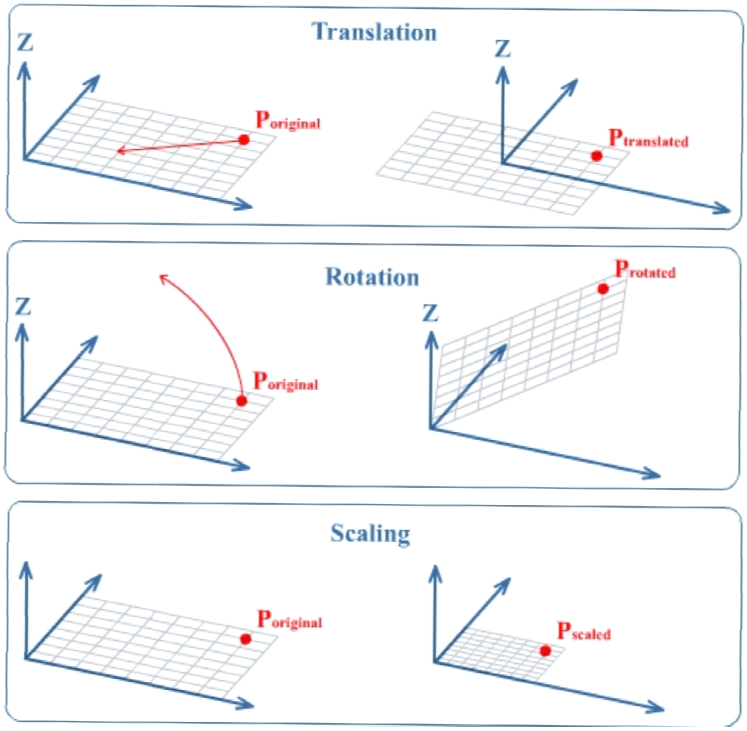


انعکاس حول هر نقطه دلخواه را نیز می توان به صورت دوران با زاویه ۱۸۰ درجه حول آن نقطه در نظر گرفت و از روش ارائه شده برای دوران بهره برد.

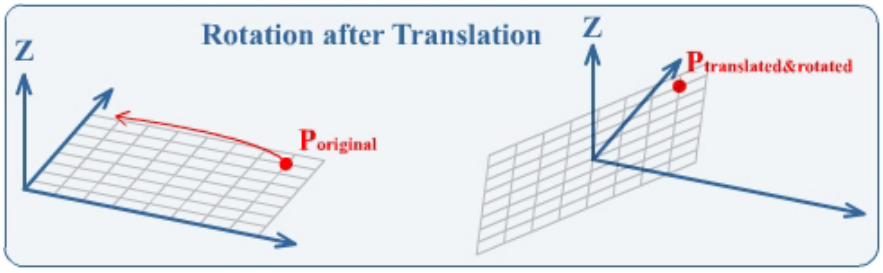


۱-۴ تبدیلات در فضای سه بعدی

تبدیل هر نقطه سه بعدی، مختصات آن را به مختصات نقطه سه بعدی دیگری تغییر می دهد. در شکل زیر سه تبدیل پایه ای نشان داده شده است:



اولین تبدیل مختصات تمام نقاط موجود در صفحه را به سمت پائین و چپ منتقل کرده است. به چنین تبدیلی انتقال[6] گفته می شود. در دومین تبدیل تمام نقاط موجود در صفحه حول محور z چرخش داده شده اند. به این نوع تبدیل دوران[7] گفته می شود. سومین تبدیل، تغییر مقیاس گفته شده و در آن مختصات تمام نقاط در عددی ضرب شده و باعث تغییر اندازه شکل می شود. این سه تبدیل تبدیلات پایه ای نامیده می شوند زیرا هر نوع تبدیلی را می-توان با استفاده از ترکیب این تبدیلات ایجاد کرد. در شکل زیر نتیجه دوران بعد از یک انتقال نشان داده شده است:



۱-۴-۱ اصول اساسی

اصل اول: برای نشان دادن مختصات یک یا مجموعه ای از نقاط می توان از ماتریسها استفاده کرد. همچنین برای انجام هر یک از تبدیلات فوق می توان از ماتریسها استفاده کرد. بنابراین می توان ماتریسهائی با نامهای Mtrans، Mrot و Mscal در نظر گرفت.

اصل دوم: اگر ماتریس تبدیلی را در ماتریس یک یا چند نقطه ضرب کنیم، ماتریس حاصل مختصات نقاط تبدیل شده را نشان خواهد داد:

$$\begin{bmatrix} Xtransformed \\ Ytransformed \\ Ztransformed \end{bmatrix} = Mtransformation * \begin{bmatrix} Xoriginal \\ Yoriginal \\ Zoriginal \end{bmatrix}$$

اصل سوم: ضرب دو ماتریس تبدیل M_1 و M_2 در یکدیگر ماتریس تبدیل M_3 را نتیجه خواهد داد که ترکیب تبدیلات M_1 و M_2 را در خود خواهد داشت. این خصوصیت بسیار سودمند می باشد، به عنوان مثال شکل فوق را در نظر بگیرید که یک عمل انتقال و سپس یک عمل دوران انجام شده است. یک راه انجام تبدیلات فوق ضرب ماتریس انتقال M_1 در ماتریس نقاط و سپس ضرب ماتریس دوران M_2 در حاصلضرب دو ماتریس قبلی است. بنابراین به ازای هر نقطه دو عمل ضرب خواهیم داشت که در رابطه زیر نشان داده شده است:

$$\begin{bmatrix} Xtran \\ Ytran \\ Ztran \end{bmatrix} = Mtranslation * \begin{bmatrix} Xoriginal \\ Yoriginal \\ Zoriginal \end{bmatrix}$$

$$\begin{bmatrix} Xtran + rot \\ Ytran + rot \\ Ztran + rot \end{bmatrix} = Mrotation * \begin{bmatrix} Xtran \\ Ytran \\ Ztran \end{bmatrix}$$

با استفاده از اصل سوم ابتدا می توان دو ماتریس تبدیل M_1 و M_2 را در یکدیگر ضرب کرده و ماتریس تبدیل M_3 را بدست آورد. سپس ماتریس نقاط را در ماتریس تبدیل M_3 ضرب کرد. روش دوم بسیار سریعتر انجام می شود زیرا ضرب دو ماتریس M_1 و M_2 بسیار سریع انجام می شود (۶۴ ضرب و ۴۸ جمع) در حالی که ضرب ماتریس نقاط صحنه در یک ماتریس تبدیل بسیار پرهزینه است (به ازای هر نقطه ۱۶ ضرب و ۱۲ جمع).

---

[6] Translation
[7] Rotation

$$Mtran+rot = Mtranslation + Mrotation$$

$$\begin{bmatrix} Xtran+rot \\ Ytran+rot \\ Ztran+rot \end{bmatrix} = Mtran+rot * \begin{bmatrix} Xoriginal \\ Yoriginal \\ Zoriginal \end{bmatrix}$$

۱-۴-۲ اعمال ریاضی ماتریسها

همانطور که می دانیم هر ماتریسی از n سطر و m ستون و n*m درایه تشکیل شده است. در حقیقت ماتریس چیزی نیست جز جدولی از اعداد. جهت پردازش نقاط سه بعدی معمولا از ماتریسهائی به شکل زیر با سه ستون و سه سطر استفاده می شود:

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

می توان مختصات یک نقطه سه بعدی را با استفاده از ماتریسی حاوی یک ستون و سه سطر (یا یک سطر و سه ستون) نشان داد. ضرب چنین ماتریسی در ماتریس تبدیل به شکل زیر خواهد بود:

$$\begin{bmatrix} Xtransformed \\ Ytransformed \\ Ztransformed \end{bmatrix} = M * \begin{bmatrix} Xoriginal \\ Yoriginal \\ Zoriginal \end{bmatrix}$$

$$\begin{bmatrix} Xtransformed \\ Ytransformed \\ Ztransformed \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} * \begin{bmatrix} Xoriginal \\ Yoriginal \\ Zoriginal \end{bmatrix}$$

$$\begin{bmatrix} Xtransformed \\ Ytransformed \\ Ztransformed \end{bmatrix} = \begin{bmatrix} m_{11} * Xoriginal + m_{12} * Yoriginal + m_{13} * Zoriginal \\ m_{21} * Xoriginal + m_{22} * Yoriginal + m_{23} * Zoriginal \\ m_{31} * Xoriginal + m_{32} * Yoriginal + m_{33} * Zoriginal \end{bmatrix}$$

مفاهیم ذکر شده را با مثالی عملی بررسی می کنیم. فرض کنید نقطه ای در فضای سه بعدی با مختصات (۶،۱۸،۹.۵) داریم. می خواهیم صحنه خود را به اندازه دو برابر بزرگنمائی کنیم. بنابراین تمام اشیا دو برابر بزرگتر خواهد شد. (دو برابر نزدیکتر به بیننده دیده می شوند). برای این منظور باید تمام نقاط را با ضریب ۲ تغییرمقیاس دهیم. در حالت کلی ماتریس تغییر مقیاس به شکل زیر است:

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{bmatrix}$$

از این رو خواهیم داشت:

$$\begin{bmatrix} Xscaled \\ Yscaled \\ Zscaled \end{bmatrix} = \begin{bmatrix} s * Xoriginal \\ s * Yoriginal \\ s * Zoriginal \end{bmatrix}$$

در مثال مطرح شده مقدار s برابر ۲ بوده و در نتیجه مختصات جدید نقطه مفروض (۱۲،۳۶،۱۹) خواهد بود.

ماتریس دوران کمی پیچیده تر از ماتریس انتقال است. سه ماتریس زیر ماتریسهای دوران یک نقطه حول محورهای x ، y و z با زاویه θ می باشد:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}, \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

به عنوان مثال فرض کنید می خواهیم نقطه ای به مختصات (۱۰،۵،۰) را به اندازه ۴۵ درجه حول محور z دوران دهیم. بنابراین ماتریس تبدیل زیر را خواهیم داشت:

$$\begin{bmatrix} \cos\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 \\ -\sin\frac{\pi}{4} & \cos\frac{\pi}{4} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cong \begin{bmatrix} 0.7071 & 0.7071 & 0 \\ -0.7071 & 0.7071 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

از آنجایی که π=3.14 و معادل با ۱۸۰ درجه است، ۴۵ درجه برابر $\pi/4$ خواهد بود. در نهایت خواهیم داشت:

$$\begin{bmatrix} Xrotated \\ Yrotated \\ Zrotated \end{bmatrix} = \begin{bmatrix} 0.7071 & 0.7071 & 0 \\ -0.7071 & 0.7071 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 10 \\ 5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.7071*10+0.7071*5 \\ -0.7071*10+0.7071*5 \\ 0 \end{bmatrix} = \begin{bmatrix} 10.6065 \\ -3.5355 \\ 0 \end{bmatrix}$$

بنابر این اگر نقطه (۱۰،۵،۰) را به اندازه ۴۵ درجه حول محور z دوران دهیم مختصات جدید نقطه برابر (۰، ۳.۵۳۵۵-، ۱۰.۶۰۶۵) خواهد بود. البته خوشبختانه تمام این محاسبات توسط خود DirectX انجام می شود. تنها مشکل باقی مانده این است که هیچ ماتریس مناسبی وجود ندارد که با ضرب ماتریس نقطه در آن، نقطه مفروض انتقال یابد. برای حل این مشکل DirectX از ماتریسهای همگن استفاده می کند.

۱-۴-۳ ماتریسهای همگن سه بعدی

اگر به ماتریسهای استفاده شده توسط DirectX توجه کنید، خواهید دید که این ماتریسها از ۴ سطر و ۴ ستون و در نتیجه ۱۶ درایه تشکیل شده اند. دلیل انتخاب ۴ سطر و ۴ ستون برای این ماتریسها، توانائی انجام عمل انتقال توسط آنها است.

جهت انتقال یک نقطه باید اعدادی را با هر یک از درایه های ماتریس آن اضافه کنید. با استفاده از ماتریسهای سه بعدی تنها می توان اعدادی را به مختصات اولیه ضرب نمود. لذا جهت اضافه نمودن اعدادی ثابت به مختصات x، y و z باید از ماتریسهای چهار بعدی استفاده نمود. بنابراین مختصات یک نقطه علاوه بر سه درایه x، y و z دارای مختصات چهارمی خواهد بود که مقدار آن همواره ۱ است. به عنوان مثال نقطه ای سه بعدی با مختصات (۱۰، ۳،۵) به صورت (۱۰، ۵، ۳، ۱) نشان داده شده و مختصات همگن آن نقطه نامیده می شود.

ماتریسهای تغییر مقیاس و دوران به شکل قبل باقی خواهند ماند با این تفاوت که درایه m_{44} در آنها ۱ و بقیه درایه های سطر و ستون چهارم صفر می باشد. در زیر ماتریس تغییرمقیاس و ماتریس دوران حول محور y نشان داده شده است:

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

با وجود اینکه این ماتریسها اندکی پیچیده تر شده اند اما حالا می توان از آنها برای انتقال نیز استفاده نمود. ماتریس انتقال را می توان به شکل زیر تعریف نمود:

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

فرض کنید می خواهیم نقطه ای با مختصات (۰، ۵، ۱۰) را به اندازه مقادیر (۴، ۲، ۸-) انتقال دهیم. خواهیم داشت:

$$\begin{bmatrix} Xtranslated \\ Ytranslated \\ Ztranslated \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -8 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 10 \\ 5 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+(-8)*1 \\ 1*5+2*1 \\ 1*0+4*1 \\ 1*1 \end{bmatrix} = \begin{bmatrix} 2 \\ 7 \\ 4 \\ 1 \end{bmatrix}$$

همانطور که انتظار می رفت مختصات جدید نقطه (۴، ۷، ۲) خواهد شد. به این ترتیب:

- اکنون ماتریسی داریم که هر یک از تبدیلات اساسی را می توان توسط آن انجام داد.
- ضرب این ماتریس در ماتریس مختصات نقطه مختصات تبدیل شده نقطه را به دست خواهد داد.
- ضرب دو ماتریس تبدیل مختلف در یکدیگر، ماتریس تبدیلی را نتیجه خواهد داد که ضرب آن در مختصات یک نقطه هر دو تبدیل را بر روی نقطه اعمال خواهد کرد.

با توجه به مطالب ذکر شده روشن است که درایه چهارم افزوده شده به مختصات نقاط به معنی بعد چهارم نبوده و اصولا هیچ معنی خاصی ندارد. تنها مزیت آن امکان تعریف ماتریس انتقال همانند ماتریسهای دوران و تغییر مقیاس است.

علاوه بر این، درایه چهارم در مختصات همگن نقطه ممکن است مقداری غیر از یک را نیز بپذیرد. طبق یک قانون کلی سه درایه اول در مختصات همگن نقطه را می توان بر درایه چهارم تقسیم کرد. حال اگر درایه چهارم ۱ باشد این تقسیم تغییری در آنها ایجاد نخواهد کرد اما در غیراین صورت مختصات همگن نقطه با مختصات واقعی آن متفاوت خواهد بود. به عنوان مثال مختصات (۲، ۰، ۱۰، ۲۰) و (۱، ۰، ۵، ۱۰) هر دو به یک معنی، و نشان دهنده نقطه (۰، ۵، ۱۰) خواهند بود. بنابر این راه حل ساده تبدیل مختصات همگن به مختصات سه بعدی به شکل زیر خواهد بود:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \Rightarrow \begin{bmatrix} X/W \\ Y/W \\ Z/W \end{bmatrix}$$

فصل دوم: برنامه نویسی DirectX

مقدمه

امروزه با ارائه کتابخانه ها و ابزارهای مختلف امکان یادگیری برنامه نویسی بازی برای هر فردی فراهم شده است. در ابتدا به صورت اجمالی به معرفی سه کتابخانه مهم گرافیکی خواهیم پرداخت و سپس مباحث عملی خود را با کتابخانه DirectX ادامه خواهیم داد. سه کتابخانه مهم در حوزه برنامه نویسی بازی عبارتند از:

- OpenGL
- Microsoft DirectX
- Microsoft XNA

- OpenGL:

شرکت سیلیکون گرافیکس کتابخانه OpenGL را با هدف ساخت یک API برای توسعه برنامه‌های گرافیکی دوبعدی و سه بعدی به صورت کدباز عرضه کرد. در اوایل پیدایش OpenGL، از آن در کارهای صنعتی، طراحی وسایل داخلی، مکانیکی و نیز در آنالیزهای علمی و آماری استفاده می‌شد اما در سال ۱۹۹۶، توسعه‌دهندگان بازیهای کامپیوتری از نسخه ویندوزی آن برای ساخت بازیهای کامپیوتری استفاده کردند.

OpenGL برای پشتیبانی از گستره وسیعی از تکنیکهای گرافیکی همچون نورپردازی، سایه سازی، حرکت محو و مدل‌سازی و... طراحی شده است. مجموعه امکانات OpenGL شبیه Direct3D است. ولی API سطح پایین‌تر آن باعث می‌شود کنترل خوبی روی عناصر اصلی ایجاد صحنه‌های سه بعدی مانند اطلاعات سه‌ضلعی‌ها داشته باشد.

OpenGL Performer، OpenGL Volumizer و OpenGL Multipipe SDK از رابطهای برنامه‌نویسی قدرتمند و کامل آن به شمار می روند. برنامه‌های کاربردی نوشته شده بر پایه OpenGL Multipipe SDK نیز به نرمی و روانی روی سیستمهای تک پردازنده‌ای و چند پردازنده‌ای با سیستمهای گرافیکی قدرتمند اجرا می‌شوند.

- DirectX:

با پیدایش سیستم عامل ویندوز، بستری مناسب برای برنامه های تجاری ایجاد شد اما این سیستم عامل به نرم افزارها اجازه دسترسی مستقیم به سخت افزار را نمی داد و نرم افزارهای چندرسانه ای همانند بازیها که نیاز به دسترسی سطح پایین به سخت افزار داشتند با مشکل مواجه شدند. توابع API ویندوز نیز نتوانستند جوابگوی نیازهای این گونه نرم افزارها باشند. از این رو شرکت مایکروسافت در سال ۱۹۹۵ برای اولین بار یک محیط برنامه نویسی جدید به نام DirectX معرفی کرد که به یک استاندارد جهت برنامه نویسی چندرسانه ای در سیستم عامل ویندوز تبدیل گردید. در واقع DirectX از یک سری توابع برنامه نویسی API تشکیل شده است که به برنامه نویس اجازه دسترسی به امکانات ویژه سخت افزارها نظیر شتاب دهنده های گرافیکی و کارتهای صدا را می دهد.

کتابخانه DirectX مجموعه ای از رابطهای برنامه نویسی کاربردی است که برای اداره کردن وظایف مربوط به برنامه های چندرسانه ای به ویژه برنامه ریزی بازی و ویدئو، از سوی شرکت مایکروسافت ارائه شده است.

DirectX در سه قالب Runtimes، Redistributable و Software Development Kit عرضه شده است که برای استفاده از کتابخانه آن جهت برنامه نویسی باید DirectX Software Development Kit بر روی سیستم نصب شود.

- XNA:

XNA مخفف Xbox New Architecture است و مجموعه ابزار ساخت بازی بر پایه دات نت است و هدف از طراحی آن ارائه مجموعه کتابخانه ها و کلاسهای وسیعی بود که توسعه دهندگان بازی بتوانند با کمترین مقدار

کدنویسی و اجرای کد در محیطی مدیریت شده، از پلتفرمهای مختلف حداکثر استفاده را ببرند. بازیهای ایجاد شده بر پایه XNA قابل اجرا بر روی پلتفرم ویندوز ایکس پی و بالاتر، و پلتفرم Xbox 360 مایکروسافت می باشد.

XNA مجموعه ای از تکنولوژیهای سطح پایین را بسته بندی کرده و خود چارچوب مسئول هماهنگ کردن تغییرات بین پلتفرمهای مختلف هنگام جابجا شدن بازی از یک پلتفرم به پلتفرم دیگر خواهد بود. این امر به توسعه دهندگان کمک می کند به جای درگیر شدن با جزئیات پلتفرم، بیشتر به محتوا و طراحی بازی خود توجه داشته باشند.

ابزارهای اصلی XNA رایگان هستند و فقط برای ویندوز تولید شده اند. XNA Game Studio محیط توسعه مجتمع برای برنامه نویسی بازی می باشد که ساختارهای لازم برای همکاری بین ایجادکنندگان محتوای بازی، برنامه نویسان، مدیران و تست کنندگان بازی را فراهم می کند و نسخه های مختلفی دارد مثلا نسخه ۲.۰ برای ویژوال استودیو ۲۰۰۵ و نسخه ۳.۰ برای ویژوال استودیو ۲۰۰۸ طراحی شده است. با نصب XNA Game Studio، قالبهای آن به ویژوال استودیو اضافه خواهد شد. علاوه بر این Windows Phone SDK نیز شامل تمام نرم افزارها و ابزارهایی است که برای ساخت بازی با XNA لازم است.

از آنجایی که این چارچوب بر پایه دات نت نوشته شده است، هر زبان دات نتی می تواند برای ساخت بازی استفاده شود.

۲-۱ شروع برنامه نویسی

جهت ساخت فضاها و بازیهای سه بعدی توسط C# ابتدا باید DirectX Software Development Kit نسخه June 2010 را بر روی کامپیوتر نصب کرده و در محیط ویژوال استودیو رفرنسهای DirectX را به پروژه اضافه نمود. از دو رفرنس زیر در تمام پروژه هایی که از مولفه های DirectX بهره می برند باید استفاده کرد:

```
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

به صورت کلی DirectX دارای منابع زیر می باشد:

Microsoft.DirectX	شامل تمام دستورات عادی و فضاهای نام برای ساخت صحنه
Microsoft.DirectX.Direct3D	شامل توابع رسمهای سه بعدی
Microsoft.DirectX.Direct3DX	شامل توابع کار با افکتها
Microsoft.DirectX.DirectDraw	شامل توابع رسم گرافیکی
Microsoft.DirectX.DirectPlay	شامل توابع کار با شبکه
Microsoft.DirectX.DirectSound	شامل توابع کار با صدا
Microsoft.DirectX.DirectInput	شامل توابع کنترل دستگاههای ورودی
Microsoft.DirectX.AudioVideoPlayback	شامل توابع عادی صدا و تصویر
Microsoft.DirectX.Diagnostics	شامل توابع خطاها

Direct3D دو حالت کاری دارد؛ حالت بلادرنگ و حالت ابقایی. حالت ابقایی شامل توابع API سطح پائین سه بعدی است و وسیله خوبی برای تولیدکنندگان بازیها و سایر برنامه های چندرسانه ای سریع الاجرا می باشد. حالت بلادرنگ یک روش غیروابسته به دستگاه جهت ارتباط برنامه ها با سخت افزار شتاب دهنده گرافیکی است. حالت ابقایی بر روی حالت بلادرنگ ایجاد شده است.

از قسمتهای پیشرفته Direct3D می توان به بافر کردن عمق، سایه گذاری تخت و گراد، منابع نور چندتایی و انواع مختلف آن، پشتیبانی کامل از متریالها و بافتها، تبدیلات و دوران، و شبیه سازی نرم افزاری قوی اشاره کرد.

۲-۱-۱ پیکربندی Device

Device رابطی بین برنامه نویس و کارت گرافیکی متصل به کامپیوترش است. این شی به برنامه نویس امکان استفاده از توانائیهای سخت افزاری کارت گرافیکی سیستم را می دهد. ابتدا متغیری به نام device از نوع Device تعریف و به کلاس فرم اضافه می کنیم. در خط بعدی، متغیری به نام pp از نوع PresentParameters تعریف شده است. این متغیر نحوه عملکرد Device را تعیین خواهد کرد.

```
private Device device;
private PresentParameters pp;
```

پیکربندی Device را می توان در هر جائی از برنامه انجام داد اما جهت حفظ شی گرا بودن برنامه، متدی به نام InitializeDevice تعریف می کنیم. حتی می توان وظیفه پیکربندی و راه اندازی Device را به یک کلاس مجزا نیز محول کرد.

```
public void InitializeDevice()
{
   pp = new PresentParameters();
   pp.Windowed = true;
   pp.SwapEffect = SwapEffect.Discard;
   device = new Device(0, DeviceType.Hardware, this,
                       CreateFlags.HardwareVertexProcessing , pp);
}
```

پارامترهای مختلفی از Device را می توان توسط PresentParameters پیکربندی کرد. به عنوان مثال در خط دوم گفته ایم که یک برنامه پنجره وار (نه FullScreen) مد نظر ماست. خط سوم به معنی عدم تمایل به تعویض صفحات است. به عبارت دیگر تنها از یک بافر برای نمایش، استفاده شده و تعویض بافری انجام نخواهد گرفت. در نتیجه تمامی ترسیمات مستقیما در Device نوشته خواهند شد.

در خط آخر Device ایجاد شده است. پارامتر 0 به معنی استفاده از اولین کارت گرافیکی موجود در سیستم است. پارمتر دوم به معنی رندر شدن تصاویر توسط سخت افزار کارت گرافیکی است. اگر کارت گرافیکی فاقد شتاب دهنده سخت افزاری باشد می توان از DeviceType.Reference که کندتر عمل می کند، استفاده کرد. پارامتر سوم بین Device و پنجره برنامه تناظر ایجاد می کند و خروجی بافر در آن منعکس خواهد شد. پارامتر چهارم مسئول رسم نقاط را مشخص می نماید. این پارامتر باعث می شود که پردازش محاسبات نقاط رسم شده نیز توسط سخت افزار کارت گرافیکی انجام شود. اگر از گزینه CreateFlags.SoftwareVertexProcessing استفاده شود این کار توسط CPU صورت خواهد گرفت. pp ایجاد شده نیز به عنوان آرگومان آخر به تابع سازنده ارسال شده است تا تنظیمات داخل آن نیز مدنظر قرار گیرد.

چنانچه بخواهیم برنامه تمام صفحه شود و رزولیشن را نیز تغییر دهیم کافی است خط دوم را حذف کرده و بعد از خط سوم، دستورات زیر را جایگزین کنیم:

```
pp.BackBufferWidth = 600;
pp.BackBufferHeight = 800;
pp.BackBufferFormat = Format.R5G6B5;
pp.Windowed = false;
```

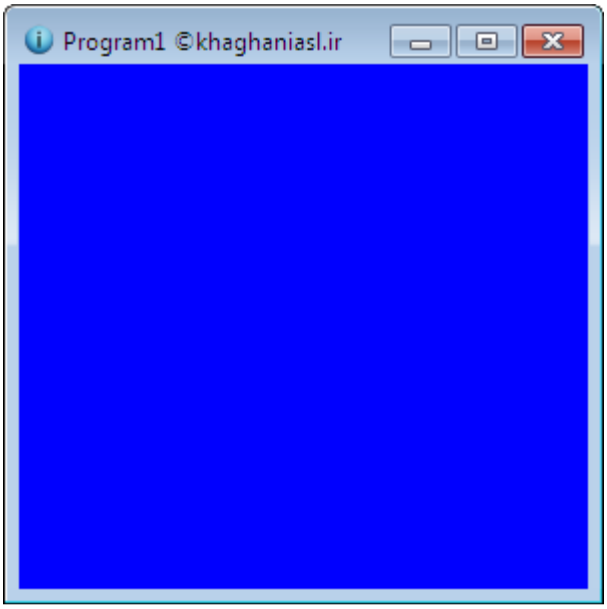
حال کافی است متد InitializeDevice در تابع سازنده فرم فراخوانی گردد:

```
public Form1()
{
   InitializeComponent();
   InitializeDevice();
}
```

البته با اجرای کد اتفاق خاصی در برنامه نیفتاده و یک پنجره خالی نشان داده خواهد شد. با سربارگذاری متد OnPaint می توان تصاویر دلخواه خود را بر روی فرم رسم کرد. برای این منظور متد زیر را به کلاس فرم اضافه می کنیم:

```
protected override void OnPaint(PaintEventArgs e)
{
   device.Clear(ClearFlags.Target, Color.Blue, 1.0f, 0);
   device.Present();
}
```

در دستور خط اول، پارامتر اول به معنی پاک کردن مولفه متناظر با Device یعنی پنجره برنامه می باشد. پارامتر دوم رنگی که صفحه را پر خواهد کرد، مشخص می‌کند. پارامتر سوم مربوط به عمق تصویر می باشد. در خط بعدی، پس از هر تغییر در Device برای مشاهده تغییرات باید متد Present را فراخوانی کنیم. حال با اجرای برنامه، رنگ پنجره به آبی تغییر خواهد کرد.



برای اینکه بتوان برنامه گرافیکی را بر روی تمام کامپیوترها اجرا کرد ابتدا بایستی مشخصات کارت گرافیکی آن سیستم را کشف کرده و بر اساس آن مشخصات، شی Device را ایجاد کرد.

قابلیتهای کارت گرافیکی توسط کلاسهای Manager، Caps و AdapterInformation قابل استخراج است. کلاس Manager مشخصات کارت گرافیکی را در قالب شی ای از نوع Caps بر می‌گرداند. مشخصات ابزارهای خروجی کارت گرافیکی را نیز در قالب شی ای از نوع AdapterInformation بر می‌گرداند.

بعد از دریافت قابلیتها کافی است شرطی قرار داده شود تا هنگام پیکربندی Device نوع پارامتر CreateFlags (HardwareVertexProcessing یا SoftwareVertexProcessing) مشخص شود:

```
Caps DeviceCapability;
DeviceCapability = Manager.GetDeviceCaps(0,DeviceType.Hardware);
CreateFlags DeviceFlags = CreateFlags.SoftwareVertexProcessing;
if(DeviceCapability.DeviceCaps.SupportsHardwareTransformAndLight)
      DeviceFlags = CreateFlags.HardwareVertexProcessing;
AdapterInformation AdapterInfo = Manager.Adapters.Default;
```

۲-۱-۲ انواع نقاط در DirectX

در DirectX به صورت کلی دو دسته نقطه داریم: Transformed و Position. نقاط Transformed تبدیل شده هستند و در مختصات دو بعدی رسم می شوند اما نقاط Position سه بعدی است.

حالتهای Transformed	
Transformed	نقطه ای که فقط مختصات دو بعدی دارد.
TransformedColored	نقطه ای که علاوه بر مختصات، رنگ دارد.
TransformedTextured	نقطه ای که علاوه بر مختصات، می توان روی آن بافت کشید.
TransformedColoredTextured	نقطه ای که علاوه بر مختصات و رنگ، می توان روی آن بافت کشید.
حالتهای Position	
PositionOnly	نقطه ای که فقط مختصات سه بعدی دارد.
PositionColored	نقطه ای که علاوه بر مختصات، رنگ دارد.
PositionTextured	نقطه ای که علاوه بر مختصات، می توان روی آن بافت کشید.
PositionColoredTextured	نقطه ای که علاوه بر مختصات و رنگ، می توان روی آن بافت کشید.
PositionNormal	نقطه ای که علاوه بر مختصات، می تواند نور را منعکس نماید.
PositionNormalColored	نقطه ای که علاوه بر مختصات و رنگ، می تواند نور را منعکس نماید.
PositionNormalTextured	نقطه ای که علاوه بر مختصات، می توان روی آن بافت کشید و می تواند نور را منعکس نماید.

در بخشهای بعدی از این نقاط استفاده خواهیم کرد.

۲-۲ ترسیم مثلث

هر شی توسط Direct3D با استفاده از سه ضلعی ها رسم می شود. هر سه ضلعی توسط سه بردار که هر یک حاوی سه مقدار x,y,z جهت تعیین مختصات نقاط آن می باشند، تعریف می شود. البته فقط داشتن مختصات نقاط کافی نیست به عنوان مثال می توان برای هر نقطه رنگی را نیز مشخص کرد.

جهت تعیین تمام این مشخصات برای یک نقطه از Vertex استفاده می شود. یک Vertex لیستی از مشخصات یک نقطه شامل موقعیت، رنگ و... می باشد. DirectX جهت پیاده سازی Vertexها کلاسی به نام CustomVertex ارائه کرده است. جهت رسم مثلث، ابتدا آرایه ای به نام vert در کلاس تعریف می کنیم:

```
private CustomVertex.TransformedColored[] vert;
```

سپس متدی به نام DrawTriangle به شکل زیر به برنامه قبلی اضافه می نمائیم:

```
public void DrawTriangle()
{
  vert = new CustomVertex.TransformedColored[3];
  vert[0].Position = new Vector4(150, 100, 0, 1);
  vert[0].Color = Color.Green.ToArgb();
  vert[1].Position = new Vector4((this.Width + this.Height)/3 ,
                                 100, 0, 1);
  vert[1].Color = Color.Yellow.ToArgb();
  vert[2].Position = new Vector4(250, 200, 0, 1);
  vert[2].Color = Color.Red.ToArgb();
}
```

اولین خط، آرایه ای از Vertexها را ایجاد می کند. استفاده از TransformedColored موجب ایجاد تطابق بین مختصات نقاط طرح، با مختصات صفحه نمایش می شود. به این ترتیب که گوشه چپ و بالای فرم به عنوان نقطه (۰،۰،۰) در نظر گرفته می شود. هر نقطه می تواند رنگ مخصوص به خود را داشته باشد. سپس مختصات و رنگ سه نقطه را مشخص کرده ایم. از آنجایی که نقاط Transformed دو بعدی است بعد z نادیده گرفته می شود. بعد چهارم نقاط نیز مقدار همگن است که در هنگام تبدیلات به کار می رود.

تنها کار باقی مانده، تحویل مثلث ایجاد شده برای رسم به device است، پس سربارگذاری متد OnPaint را به شکل زیر تغییر می دهیم:

```
protected override void OnPaint(PaintEventArgs e)
{
  device.Clear(ClearFlags.Target,Color.Blue, 0, 1);
  device.BeginScene();
  device.VertexFormat = CustomVertex.TransformedColored.Format;
  device.DrawUserPrimitives(PrimitiveType.TriangleList, 1, vert);
  device.EndScene();
  device.Present();
}
```

device.BeginScene() به معنی ایجاد یک صحنه جدید در Device است. یک صحنه؛ جهانی یکپارچه با تمام اشیا داخل آن است. device.EndScene() به معنی پایان ساخت صحنه است. تمام دستورات ساخت یک صحنه باید بین این دو فراخوانی درج شوند. هنگام ترسیم صحنه باید به DirectX اطلاع دهیم که در حال ترسیم صحنه هستیم و با بافر کاری نداریم.

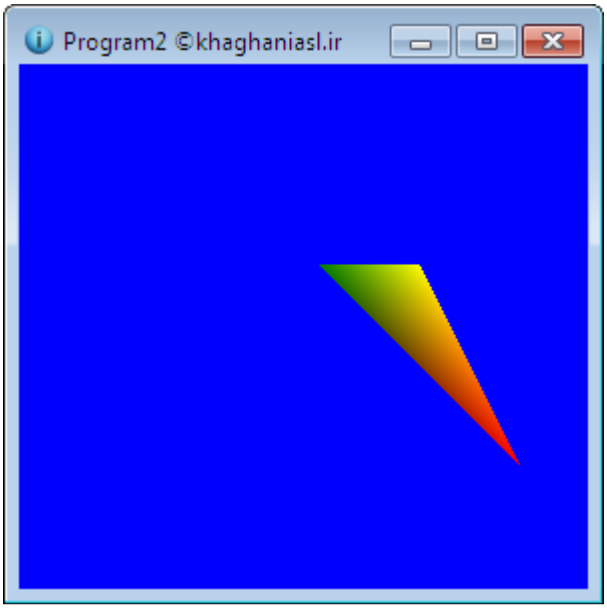
خطوط سوم و چهارم این متد، صحنه ما را که فقط شامل یک مثلث ساده است، ایجاد می کند. در خط سوم ابتدا مشخص کرده ایم که چه نوع Vertexای استفاده کرده ایم. خط چهارم سبب رسم مثلث خواهد شد. در این خط، پارامتر اول مشخص می کند که لیستی از مثلثهای مجزا رسم خواهد شد. پارامتر دوم تعداد مثلثهائی که باید رسم شود را مشخص می کند. در صورتی که بخواهیم ۳ مثلث رسم کنیم به آرایه ای ۹ تائی از نقاط نیاز خواهیم داشت. پارامتر سوم نیز آرایه ای که مختصات مثلث در آنجا تعریف شده است را مشخص می‌نماید.

دستورات فوق باعث رسم مثلث در بافر کارت گرافیکی خواهد شد اما تا زمانی که متد Present از device فراخوانی نشود، محتویات بافر بر روی مانیتور ظاهر نخواهد شد.

حال کافی است متد DrawTriangle() در تابع سازنده فرم فراخوانی گردد.

```
public Form1()
{
   InitializeComponent();
   InitializeDevice();
   DrawTriangle();
}
```

اجرای کدهای فوق باعث رسم یک مثلث رنگی بر روی پنجره آبی خواهد شد:



جهت ثابت ماندن شکل مثلث هنگام تغییر اندازه فرم، کد زیر را می توان به انتهای متد OnPaint اضافه کرد:

```
this.Invalidate();
```

این دستور باعث صدور یک پیام ترسیم به پنجره برنامه شده و در نتیجه دوباره متد OnPaint فراخوانی خواهد شد. به این ترتیب هر تغییر در device بلافاصله در پنجره مشاهده خواهد شد. (البته در صورت Present کردن)
می توان کد زیر را نیز به متد OnPaint اضافه کنیم:

```
this.SetStyle(ControlStyles.ResizeRedraw | ControlStyles.Opaque,
              true);
```

این دستور، پنجره را مجاب می کند در صورت دریافت پیام ترسیم، فقط محتویات پنجره (نه پس زمینه) را مجددا رسم نماید. پارامتر اول بدین منظور است که اگر کاربر اندازه پنجره را تغییر دهد مجددا متد OnPaint فراخوانی شود. پارامتر دوم بدین معنا است که نمی خواهیم پس زمینه مجددا رسم شود و پارامتر سوم یعنی این تنظیمات را هم اکنون روی فرم اعمال نماید.

۳-۲ دوربین و مختصات فضای جهانی

در بخش قبل، یک مثلث با استفاده از مختصات Transformed رسم کردیم. ویژگی نقاط Transformed این است که مختصات قطعی آنها را باید در صفحه نمایش مشخص کرد. اما معمولا لازم است از مختصات UnTransformed استفاده شود. با استفاده از این نوع می توان یک محیط کاملا سه بعدی و یکپارچه ایجاد کرده و اشیا سه بعدی را در فضاهای دلخواه در محیط قرار داد. همچنین می توان مختصات دوربین (ناظر) و زاویه دید آن را مشخص کرد.

برای شروع، مختصات نقاط مثلثمان را به مختصات جهانی تغییر می دهیم:

```
private CustomVertex.PositionColored[] vert;
```

متد DrawTriangle را نیز به شکل زیر تغییر می دهیم:

```
public void DrawTriangle()
{
  vert = new  CustomVertex.PositionColored[3];
  vert[0].Position = new Vector3(0, 0, 0);
  vert[0].Color = Color.Green.ToArgb();
```

```
  vert[1].Position = new Vector3(10, 0, 0);
  vert[1].Color = Color.Red.ToArgb();
  vert[2].Position = new Vector3(5,10, 0);
  vert[2].Color = Color.Yellow.ToArgb();
}
```

ابتدا نوع داده TransformedColored را به PositionColored تغییر داده ایم. سپس مختصات x,y,z هر راس از مثلث را به کمک نوع داده Vector3 تعیین کرده ایم. بدیهی است تغییر فرمت نقاط مثلث را باید توسط دستوری مشابه زیر، به اطلاع device برسانیم:

```
device.VertexFormat = CustomVertex.PositionColored.Format;
```

البته با اجرای کد فوق مثلث دوباره ناپدید خواهد شد. دلیل این اتفاق، عدم تعیین مختصات دوربین و محل نگرش آن است. برای تعیین محل دوربین، خطوط زیر را به اولین خط متد OnPaint اضافه می کنیم:

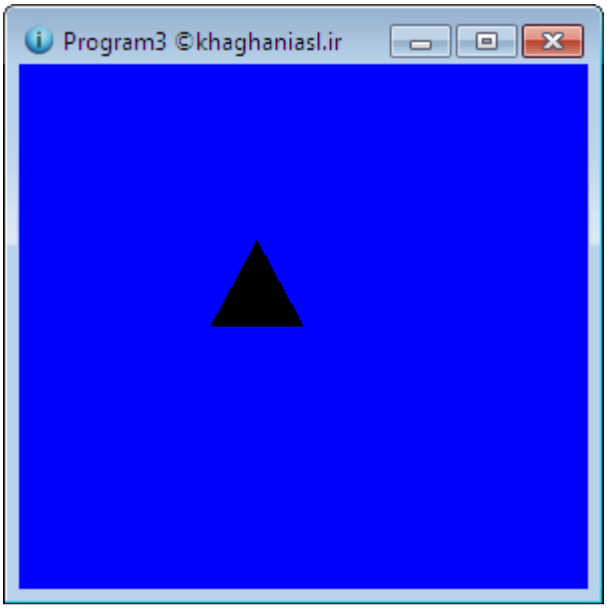
```
device.Transform.Projection =
Matrix.PerspectiveFovLH((float)Math.PI/2, this.Width/this.Height,
1f, 50f);
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0,30), new
Vector3(0,0,0), new Vector3(0,1,0));
```

خط اول، نحوه نگرش دوربین به صحنه را تعریف می نماید. برای تعیین نحوه نگرش دوربین خصوصیت Projection از device را باید پیکربندی کنیم. اولین پارامتر زاویه دید را مشخص می کند. در مثال ما این مقدار ۹۰درجه است. دومین پارامتر نسبت طول به عرض صحنه را مشخص می‌کند. این مقدار در مثال ما که از یک پنجره مربع شکل استفاده می کنیم برابر ۱ است. اگر به جای مربع از یک پنجره مستطیلی شکل استفاده می کردیم این مقدار تغییر می‌کرد. دو پارامتر آخر دامنه دید را مشخص می کنند. هر شی ای که فاصله اش نسبت به دوربین کمتر از 1f و بیشتر از 50f باشد دیده نخواهد شد. فاصله بین این دو عدد در اصطلاح Clipping Plane نامیده می‌شود زیرا هر شی، مابین این دو به صورت کلیپ، نمایش می یابد. هر چه این پارامتر بزرگ انتخاب شود، برنامه سخت تر رندر می شود.

در دومین خط، موقعیت دوربین و سمت نگرش آن در صحنه مشخص می شود. اولین پارامتر، موقعیت دوربین در صحنه را مشخص می کند. در مثال ما دوربین در ۳۰ واحد جلوتر نسبت به مرکز صحنه قرار داده می شود. دومین پارامتر نقطه ای را که دوربین به آنجا می نگرد را مشخص می کند. در مثال فوق، دوربین دقیقا به مرکز صحنه نگاه می کند. توسط این دو پارامتر مسیر نگرش دوربین مشخص می شود. حتی می توان دوربین را حول این مسیر چرخش داد. (همانند دوربین عکاسی که به طرف شخصی گرفته و سپس آن را دوران دهیم. برای این منظور باید مشخص کرد که کدامیک از بردارها باید به عنوان محور عمود در نظر گرفته شود.)

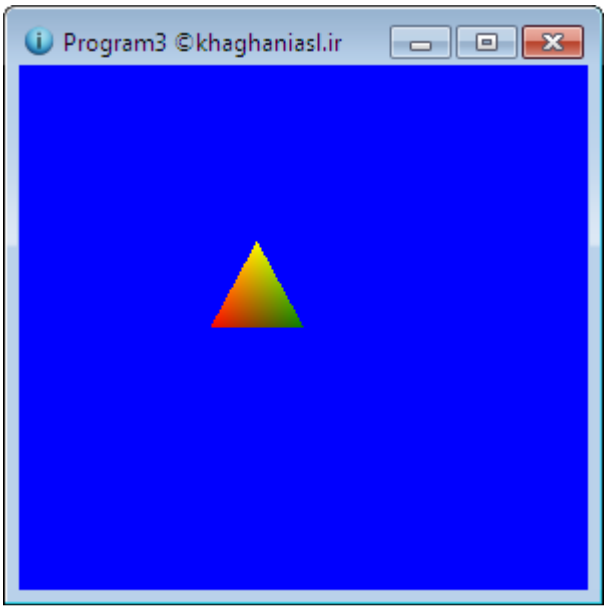
متد OnPaint به شکل زیر تغییر کرد. با اجرای کدهای نوشته شده، مثلثی کاملا سیاه را مشاهده خواهید کرد.

```
protected override void OnPaint(PaintEventArgs e)
{
  device.Transform.Projection = Matrix.PerspectiveFovLH
((float)Math.PI/2, this.Width/this.Height, 1f, 50f);
  device.Transform.View = Matrix.LookAtLH(new Vector3(0,0,30),
new Vector3(0,0,0), new Vector3(0,1,0));
  device.Clear(ClearFlags.Target,Color.Blue, 0, 1);
  device.BeginScene();
  device.VertexFormat = CustomVertex.PositionColored.Format;
  device.DrawUserPrimitives(PrimitiveType.TriangleList, 1, vert);
  device.EndScene();
  device.Present();
}
```



دلیل این امر، استفاده از فضای جهانی تاریک می باشد. در فضای جهانی، استفاده از نور برای نمایش اشیا ضروری است. استفاده از نور بعدا مفصلا بررسی می شود و فعلا در اینجا به device اعلام می کنیم که انتظار هیچ نوری را نداشته باشد. با اضافه کردن کد زیر به بعد از دستورات تعریف دوربین، دوباره مثلث رنگی خود را مشاهده خواهیم کرد:

```
device.RenderState.Lighting = false;
```



حال تمام ملزومات لازم برای استفاده از فضای جهانی آماده شده است و فقط یک نکته باقی مانده است. DirectX تنها مثلثهائی که رو در روی دوربین به صورت ساعتگرد تعریف شده اند را رسم می کند. طبق تعریف DirectX، مثلثهای رو در روی دوربین، به صورت ساعتگرد نسبت به آن رسم می شوند. اگر دوربین را در موقعیت منفی محور Zها قرار دهیم، ترتیب مختصات تعریف شده برای مثلث به صورت پادساعتگرد نسبت به دوربین قرار گرفته و رسم نمی شود. یک راه حل برای این مشکل تعریف مجدد رئوس مثلث به صورت ساعتگرد است. روش دوم درج دستور زیر بعد از تعریف دوربین است:

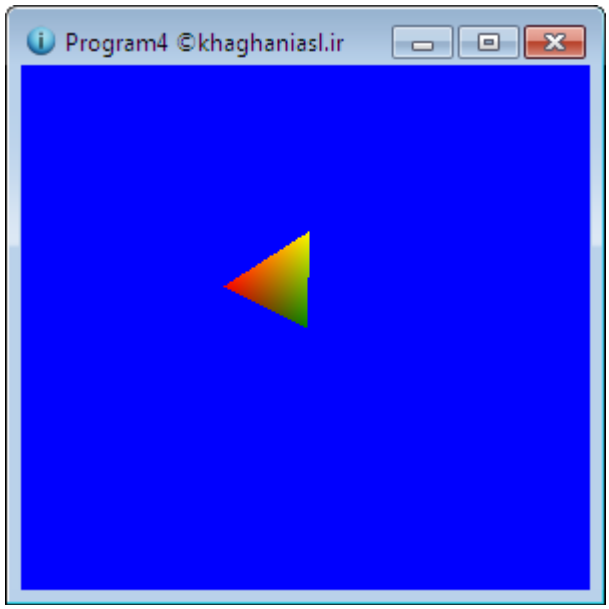
```
device.RenderState.CullMode = Cull.None;
```

این کد به سادگی باعث رسم همه مثلثها (حتی آنهایی که رو در روی دوربین نیستند) می شود. باید توجه داشت که نباید در محصول نهائی از این ویژگی استفاده کرد زیرا رسم تمام مثلثها به این شکل باعث کاهش شدید کارایی خواهد شد. با این حال هنگام طراحی می توان با خاموش کردن Culling تمام اشکال رسم شده را مشاهده کرد. دلیل استفاده از رنگ پشت زمینه آبی در این مثال، توانائی مشاهده مثلث در محیط تاریک بود. بنابر این بهتر است هنگام طراحی، یک رنگ غیرسیاه برای پشت زمینه انتخاب کنید.

۲-۳-۱ دوران و انتقال

در این بخش در مورد چرخش و انتقال مثلث صحبت خواهد شد. با توجه به اینکه از مختصات جهانی استفاده می کنیم این کار بسیار ساده است. در DirectX تمام چرخشها توسط کلاسهای Transformation انجام می گیرد و تنها کاری که برنامه نویس باید انجام دهد، تعیین محور و زاویه چرخش است. به عنوان مثال با افزودن کد زیر قبل از دستور رسم مثلث یعنی device.DrawUserPrimitives، مثلث (و تمام اشیا دیگر در آن جهان در صورت وجود) به اندازه ۰.۵ درجه حول محور Zها چرخش خواهد داشت:

```
device.Transform.World = Matrix.RotationZ(0.5f);
```



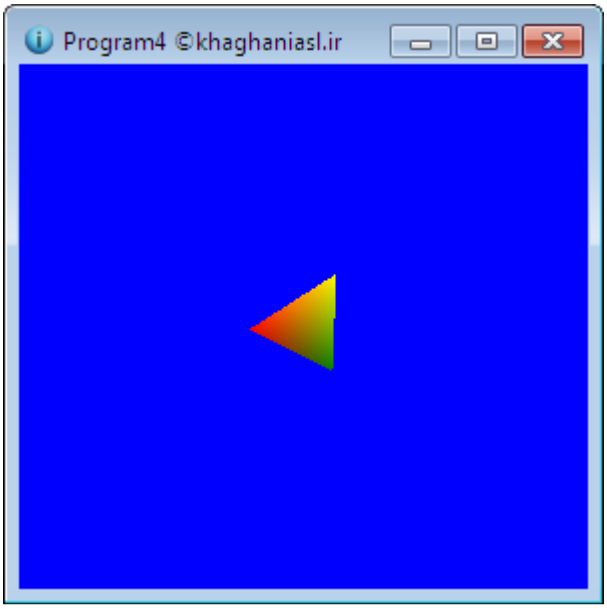
افزایش مقدار درجه باعث می شود در رسم بعدی، مثلث نسبت به موقعیت قبلی اش دوباره دوران داشته باشد. از آنجایی که یکی از گوشه های مثلث دقیقا در وسط جهان قرار دارد، بنابر این مثلث از این گوشه نسبت به محور Zها دوران می کند. اما اگر بخواهیم مثلث نسبت مرکز خودش دوران داشته باشد چه باید کرد؟

یک راه حل، تغییر مختصات گوشه های مثلث به گونه ای که وسط مثلث منطبق بر وسط جهان شود. راه حل بهتر انتقال مثلث به سمت پائین و راست[8] و سپس دوران آن است. برای این منظور ماتریس جهان جدید خود را باید در ماتریس انتقال مناسبی ضرب نمائیم:

```
device.Transform.World = Matrix.Translation(-5, -10*1/3, 0)*
Matrix.RotationZ(0.5f);
```

با اجرای برنامه خواهیم دید که مثلث حول مرکز خود دوران می کند:

[8] Translation

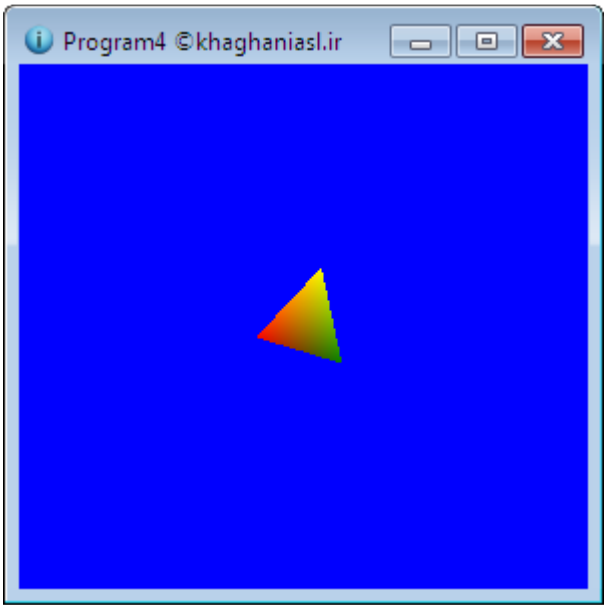


دقت کنید که ترتیب ضرب ماتریسها اهمیت دارد. به عنوان مثال اگر محل ماتریسهای دوران و انتقال را در دو طرف عملگر ضرب عوض کنیم مثلث به سمت پائین و راست انتقال می یابد ولی در محل جدید حول یکی از راسها (مانند حالت اولیه اش) دوران می یابد. می توان از محورهای دیگر نیز جهت دوران استفاده کرد. برای این منظور کافی است آخرین حرف متد دوران را برابر محور مورد نظر قرار داد.

روش پیچیده تر دیگری نیز وجود دارد که امکان ایجاد دورانهای دلخواه برای محیط را مهیا می کند. این روش Matrix.RotationAxis نام دارد. دستور زیر را در نظر بگیرید:

```
device.Transform.World = Matrix.Translation(-5, -10*1/3, 0) *
Matrix.RotationAxis(new Vector3(0.5f * 4, 0.5f * 2, 0.5f * 3),
0.5f);
```

این دستور مثلث را حول هر سه محور با زاویه های مختلف دوران می دهد.

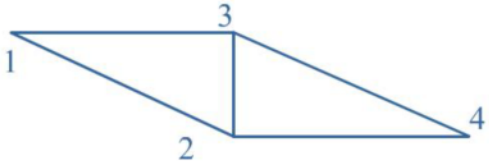


قبل از پایان این بخش به نکته کوچکی جهت افزایش کارائی برنامه می پردازیم. از آنجایی که دستورات مربوط به محل قرارگیری دوربین و تعریف رئوس مثلث همواره ثابت می باشند برای اینکه متد OnPaint سریعتر و بهتر کار

کند، می توان آنها را در دو متد مختلف مثلا به نامهای Camera و VertexDeclaration قرار داده و در تابع سازنده فراخوانی کرد.

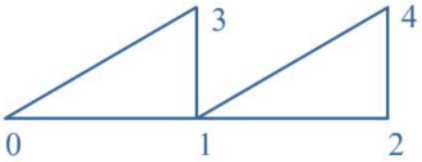
۲-۴ ترکیب رئوس با استفاده از اندیس‌ها

کار با مثلثها ساده و زیباست اما اگر تعداد زیادی مثلث داشته باشیم وضعیت چگونه خواهد بود؟ برای ایجاد سطوح مسطح و یا ناهموار بزرگ، باید تعداد زیادی مثلث در کنار هم ایجاد شوند. بسیاری از نقاط این مثلثها بر روی هم قرار می گیرد که می توان یک نقطه مشترک برای آنها در نظر گرفت. به عنوان مثال شکل زیر را در نظر بگیرید:



از ۶ راس دو مثلث فوق فقط ۴ تای آنها یکتا بوده و ۲ تای دیگر صرفا سرباری برای کارت گرافیکی است. برای رسم شکل فوق بهتر است از آرایه ای با ۴ خانه استفاده کنیم که اندیسهای ۰،۱و۲ برای مثلث اول و اندیسهای ۱،۲و۳ برای مثلث دوم باشند. به این ترتیب در رسم اشکال پیچیده تر نیازی به ذخیره سازی مجدد رئوس نخواهد بود. این ایده اساس کار VertexBuffer و IndexBuffer است.

فرض کنید می خواهیم شکل زیر را رسم کنیم:



با وجود اینکه ۶ راس داریم، استفاده از ۵ راس نیز کفایت می نماید. بنابراین متد DrawTriangle را با متد جدیدی به نام VertexDeclaration به شکل زیر جایگزین می کنیم:

```
private void VertexDeclaration()
{
  vb = new VertexBuffer(typeof(CustomVertex.PositionColored), 5,
       device, Usage.Dynamic | Usage.WriteOnly,
       CustomVertex.PositionColored.Format, Pool.Default);
  vert = new CustomVertex.PositionColored[5];
  vert[0].Position = new Vector3(0f, 0f, 0f);
  vert[0].Color = Color.White.ToArgb();
  vert[1].Position = new Vector3(5f, 0f, 0f);
  vert[1].Color = Color.White.ToArgb();
  vert[2].Position = new Vector3(10f, 0f, 0f);
  vert[2].Color = Color.White.ToArgb();
  vert[3].Position = new Vector3(5f, 5f, 0f);
  vert[3].Color = Color.White.ToArgb();
  vert[4].Position = new Vector3(10f, 5f, 0f);
  vert[4].Color = Color.White.ToArgb();
  vb.SetData(vert, 0, LockFlags.None);
}
```

در اولین خط یک VertexBuffer جهت نگهداری و مرتبط نمودن رئوس شکل تعریف شده است. اولین پارامتر نوع نقاطی را که بعدا در VertexBuffer قرار داده خواهد شد را مشخص می کند. پارامتر دوم؛ تعداد نقاط، پارامتر سوم device و پارامتر چهارم به بعد نیز برخی تنظیمات پیش فرض را مشخص می نماید. در ادامه نیز ۵ راس برای نشان دادن رئوس شکل فوق تعریف شده است. آخرین دستور نیز آرایه نقاط را در VertexBuffer قرار می دهد.

بدیهی است قبل از استفاده از متغیر vb باید آن را در کلاس تعریف نمود:

```
private VertexBuffer vb;
```

آرایه ای برای اندیسها و یک IndexBuffer برای این آرایه نیز باید در کلاس تعریف شود:

```
private int[] indices;
private IndexBuffer ib;
```

در ادامه، متدی به نام IndicesDeclaration به شکل زیر تعریف می کنیم:

```
private void IndicesDeclaration()
{
  ib = new IndexBuffer(typeof(int), 6, device, Usage.WriteOnly,
                       Pool.Default);
  indices = new int[6];
  indices[0] = 3;
  indices[1] = 1;
  indices[2] = 0;
  indices[3] = 4;
  indices[4] = 2;
  indices[5] = 1;
  ib.SetData(indices, 0, LockFlags.None);
}
```

در اولین خط این متد، یک IndexBuffer جهت نگهداری اندیس نقاط ایجاد می شود. پارامتر اول نوع نقاط، پارامتر دوم تعداد اندیسها، پارامتر سوم خود device، پارامتر چهارم نحوه تخصیص حافظه (اندیسها یکبار ساخته شده و تغییر نخواهند کرد) و پارامتر پنجم نحوه صف بندی نقاط در داخل بافر را مشخص می کند.

از آنجایی که دو مثلث رسم خواهد شد، ۶ اندیس برای IndexBuffer لازم است. در دستورات بعدی آرایه ای ایجاد شده و اندیس نقاط در آن قرار داده می شود. راس ۱ در هر دو مثلث استفاده شده، بنابراین دو بار به عنوان اندیس ظاهر شده است. البته در این مثال میزان صرفه جوئی در نقاط کم است اما در برنامه های واقعی این میزان قابل توجه خواهد بود. همچنین توجه نمائید که رئوس مثلث به صورت ساعتگرد تعریف شده است بنابراین DirectX آنها را مقابل دوربین خواهد دید. آخرین خط نیز آرایه را در بافر قرار می دهد.

حال تنها کار باقیمانده رسم مثلثها با استفاده از بافر است. پس تغییرات زیر را در متد OnPaint اعمال می کنیم:

```
protected override void OnPaint(PaintEventArgs e)
{
  device.Clear(ClearFlags.Target,Color.Blue, 0, 1);
  device.BeginScene();
  device.VertexFormat = CustomVertex.PositionColored.Format;
  device.SetStreamSource(0, vb, 0);
  device.Indices = ib;
  device.DrawIndexedPrimitives(PrimitiveType.TriangleList,
                               0, 0, 5, 0, 2);
  device.EndScene();
  device.Present();
}
```

بعد از ایجاد صحنه، تعیین فرمت نقاط همچنان باید انجام شود. در دستورات بعدی VertexBuffer و IndexBuffer به عنوان منبع داده ها برای device مشخص شده اند. در نهایت تصویر با استفاده از متد DrawIndexedPrimitives رسم شده است. در این متد اولین پارامتر، همچنان نشان دهنده لیستی از مثلثها جهت رسم است. دومین پارامتر، اندیس شروع داده‌ها در IndexBuffer را مشخص می‌کند. پارامتر سوم کوچکترین عدد موجود در IndexBuffer را مشخص می کند. اگر مقدار این پارامتر؛ صفر در نظر گرفته شود DirectX به

درستی شکل را رسم می کند اما سرعت پردازش کندتر خواهد شد. پارامتر چهارم تعداد نقاط و پارامتر پنجم اندیس شروع را مشخص می کند. در نهایت تعداد مثلثهائی که باید رسم شود را مشخص می کنیم.

جهت افزایش کارایی و سرعت بخشیدن به اجرای متد OnPaint، کدهای مربوط به تعیین مختصات دوربین و محل نگرش آن را نیز در متدی به نام Camera به شکل زیر قرار می دهیم:

```
private void Camera()
{
  device.Transform.Projection = Matrix.PerspectiveFovLH(
          (float)Math.PI/4, this.Width/this.Height, 1f, 50f);
  device.Transform.View = Matrix.LookAtLH(new Vector3(0, 0, -30),
                      new Vector3(0, 0, 0), new Vector3(0, 1, 0));
  device.RenderState.Lighting = false;
}
```

حال با فراخوانی متدهای جدید در تابع سازنده فرم، پس از اجرای برنامه، دو مثلث سفید متوالی بر روی فرم دیده خواهد شد:

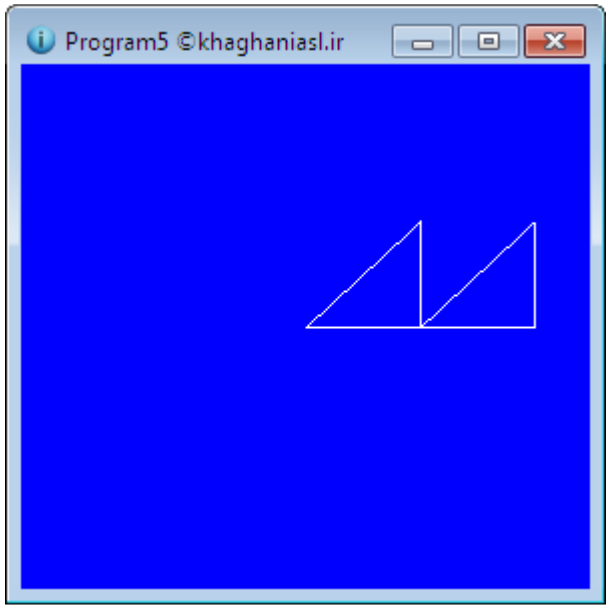
```
public Form1()
{
  InitializeComponent();
  InitializeDevice();
  Camera();
  VertexDeclaration();
  IndicesDeclaration();
}
```

چنانچه کارت گرافیکی سیستم دارای شتاب دهنده گرافیکی نباشد، اجرای کدهای فوق، منجر به خروجی تهی خواهد شد و باید از CPU برای کارهای گرافیکی استفاده کرد. بنابراین تغییراتی نیز در متد InitializeDevice برنامه بصورت زیر بایستی اعمال شود:

```
public void InitializeDevice()
{
  pp = new PresentParameters();
  pp.Windowed = true;
  pp.SwapEffect = SwapEffect.Discard;
  device = new Device(0, DeviceType.Reference, this,
                      CreateFlags.SoftwareVertexProcessing , pp);
}
```

اگر بخواهیم فقط خطوط مثلث دیده شوند، کافی است در متد InitializeDevice، پس از ایجاد device دستور زیر را اضافه نمائیم:

```
device.RenderState.FillMode = FillMode.WireFrame;
```



۲-۵ ایجاد زمین

۲-۵-۱ ایجاد زمین با استفاده از مثلث

با توجه به مطالب قبلی، اکنون می توان با اتصال رئوس مثلثها، یک زمین ایجاد کرد. فرض کنید زمینی دارای ۳*۴ نقطه می باشد. (البته بعدا زمین پهناورتری را ایجاد خواهیم کرد) پس ابتدا دو متغیر زیر را به کلاس اضافه می‌کنیم:

```
private int Height = 3;
private int Width = 4;
```

آرایه ای دو بعدی نیز در ابتدای کلاس می سازیم تا اطلاعات مربوط به نقاط را در آن نگهداری کنیم. البته هنوز مقداری برای مختصات Z نقاط مشخص نکرده ایم.

```
private int[,] Points;
```

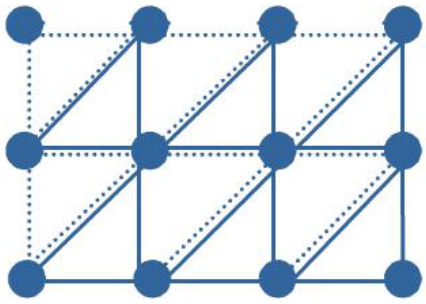
متدی به نام FillPoints نیز جهت پر کردن آرایه فوق تعریف می کنیم:

```
private void FillPoints()
{
  Points = new int[4,3];
  Points[0,0]= 0;
  Points[1,0]= 0;
  Points[2,0]= 0;
  Points[3,0]= 0;
  Points[0,1]= 1;
  Points[1,1]= 0;
  Points[2,1]= 2;
  Points[3,1]= 2;
  Points[0,2]= 2;
  Points[1,2]= 2;
  Points[2,2]= 4;
  Points[3,2]= 2;
}
```

پس از بارگذاری اطلاعات آرایه، نوبت به ایجاد نقاط می رسد. از آنجایی که زمینی با ابعاد ۴*۳ تعریف کرده ایم بنابراین ۱۲ نقطه خواهیم داشت. فاصله نقاط نسبت به یکدیگر ثابت است، پس به سادگی می توان کدهای مربوط به متد VertexDeclaration در برنامه قبل را به شکل زیر تغییر داد:

```
private void VertexDeclaration()
{
  vb = new VertexBuffer(typeof(CustomVertex.PositionColored),
         Width * Height, device, Usage.Dynamic | Usage.WriteOnly,
         CustomVertex.PositionColored.Format, Pool.Default);
  vert = new CustomVertex.PositionColored[Width * Height];
  for (int x = 0; x < Width; x++)
  {
    for (int y = 0; y < Height; y++)
    {
      vert[x + y * Width].Position = new Vector3(x, y, 0);
      vert[x + y * Width].Color = Color.White.ToArgb();
    }
  }
  vb.SetData(vert, 0, LockFlags.None);
}
```

کد فوق صرفا ۱۲ نقطه به رنگ سفید تعریف می نماید. برای تعریف مثلثها، بایستی این نقاط به یکدیگر متصل شوند. بدیهی است این کار با اندیس گذاری روی نقاط انجام خواهد شد. بهترین روش برای انجام این کار تعریف دو دسته از نقاط به شکل زیر است:



برای رسم مثلثهائی با خطوط ضخیم تر، متد IndicesDeclaration قبلی را به شکل زیر تغییر می دهیم:

```
private void IndicesDeclaration()
{
  ib = new IndexBuffer(typeof(int), (Width - 1)*(Height - 1)*3,
                       device, Usage.WriteOnly, Pool.Default);
  indices = new int[(Width - 1) * (Height - 1) * 3];
  for (int x = 0; x < Width - 1; x++)
  {
    for (int y = 0; y < Height - 1; y++)
    {
      indices[(x + y *(Width-1))*3] = (x+1) + (y+1) * Width;
      indices[(x + y *(Width-1))*3 + 1] = (x+1) + y * Width;
      indices[(x + y *(Width-1))*3 + 2] = x + y * Width;
    }
  }
  ib.SetData(indices, 0, LockFlags.None);
}
```

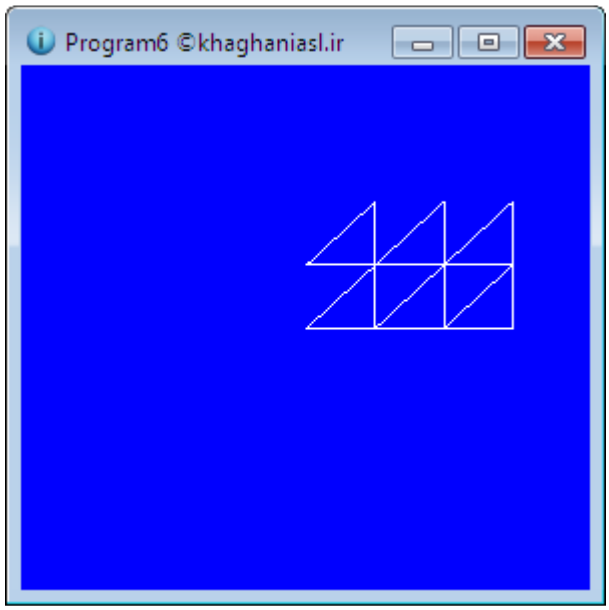
با توجه به شکل، دو ردیف حاوی ۳ مثلث داریم و در کل ۶ مثلث باید رسم شود. بنابراین ۱۸=۶*۳ اندیس لازم خواهد بود. ساختار مورد نیاز برای این مثلثها در دو خط اول ایجاد شده است. در خطوط بعدی مثلثها با پیمایش

اندیسهای x و y ایجاد شده اند. هر مثلث به ۳ اندیس نیاز دارد. از این رو اندیسها در ۳ ضرب شده اند. با توجه به مفهوم Culling، مثلثها ساعتگرد رسم می شوند. به همین دلیل ابتدا رئوس گوشه بالای سمت راست مثلثها، سپس گوشه سمت راست پائین و در نهایت گوشه سمت چپ پائین مثلثها مشخص شده اند.

تنها کار باقی مانده رسم مثلثها است. پس متد OnPaint را کمی تغییر می دهیم:

```
protected override void OnPaint(PaintEventArgs e)
{
  device.Clear(ClearFlags.Target,Color.Blue, 0, 1);
  device.BeginScene();
  device.VertexFormat = CustomVertex.PositionColored.Format;
  device.SetStreamSource(0, vb, 0);
  device.Indices = ib;
  device.DrawIndexedPrimitives(PrimitiveType.TriangleList,
                   0, 0, Width * Height, 0, indices.Length / 3);
  device.EndScene();
  device.Present();
}
```

اگر دوربین را در موقعیت (۱۰-،۰،۰) قرار داده و برنامه را اجرا کنید، ۶ مثلث مشاهده خواهید کرد:



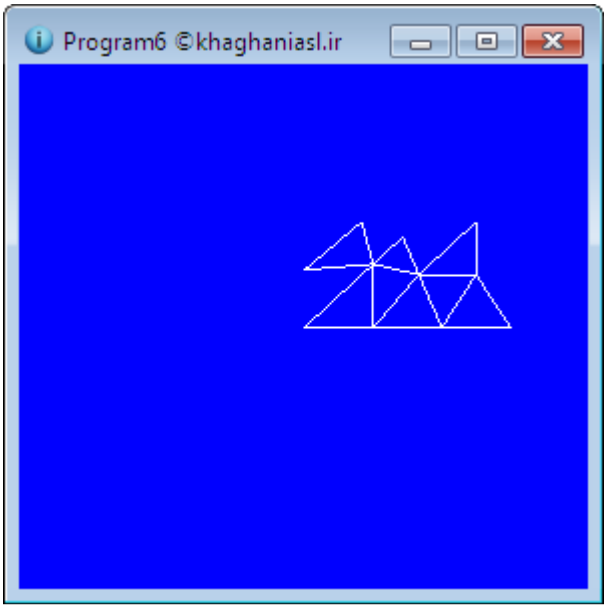
مختصات Z همه نقاط در تمام مثلثها با هم برابر است. حال ارتفاع نقاط را بر اساس آرایه Points تعریف شده، در خط هفتم متد VertexDeclaration به شکل زیر تغییر می دهیم:

```
vert[x + y * Width].Position = new Vector3(x,y, PointsData[x,y]);
```

متد FillPoints را نیز در تابع سازنده فرم قبل از متد VertexDeclaration فراخوانی می نمائیم:

```
public Form1()
{
  InitializeComponent();
  InitializeDevice();
  Camera();
  FillPoints();
  VertexDeclaration();
  IndicesDeclaration();
}
```

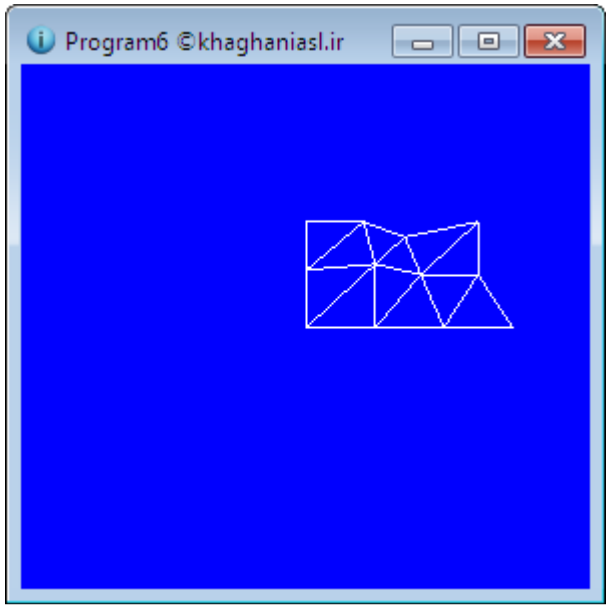
با اجرای برنامه مشاهده خواهیم کرد که راس بالائی مثلثها اندکی به سمت جلو منحرف شده است:



اکنون زمان رسم دسته دوم مثلثها است. برای این کار از همان رئوس قبلی استفاده خواهیم کرد و تنها کاری که باید انجام دهیم افزودن اندیسهای جدید است. از آنجایی که مثلثهای دسته دوم نیز باید نسبت به دوربین ساعتگرد رسم شوند ضریب ۳ با ضریب ۶ عوض خواهد شد. بنابراین متد IndicesDeclaration را به شکل زیر تغییر می-دهیم:

```
private void IndicesDeclaration()
{
  ib = new IndexBuffer(typeof(int), (Width-1) * (Height-1) * 6,
                       device, Usage.WriteOnly, Pool.Default);
  indices = new int[(Width - 1) * (Height - 1) * 6];
  for (int x = 0; x < Width - 1; x++)
  {
    for (int y = 0; y < Height - 1; y++)
    {
      indices[(x + y * (Width-1)) * 6] = (x+1) + (y+1) * Width;
      indices[(x + y * (Width-1)) * 6 + 1] = (x+1) + y * Width;
      indices[(x + y * (Width-1)) * 6 + 2] = x + y * Width;
      indices[(x + y * (Width-1)) * 6 + 3] = (x+1)+ (y+1)* Width;
      indices[(x + y * (Width-1)) * 6 + 4] = x + y * Width;
      indices[(x + y * (Width-1)) * 6 + 5] = x + (y+1) * Width;
    }
  }
  ib.SetData(indices, 0, LockFlags.None);
}
```

با اجرای برنامه، زمین کوچکی ظاهر خواهد شد که برای تغییر اندازه آن، تنها کافی است مختصات Width و Height را به همراه آرایه Points عوض کنیم.



۲-۵-۲ ایجاد زمین از روی فایل raw

برای ایجاد پستی و بلندی در صحنه (تغییر ارتفاع نقاط) دو روش کلی وجود دارد: استفاده از توابع ریاضی و استفاده از الگوهای تصویری. می توان ناهمواریهای دلخواه بر روی سطح ایجاد شده را با استفاده از الگوهای مختلف مانند تصاویر ایجاد کرد.

یک فایل raw تصویری سیاه و سفید را می تواند نگهداری نماید به طوری که برای هر پیکسل تصویر، یک بایت درون فایل ذخیره می شود. در ضمن اطلاعات پیکسلها به صورت عکس ذخیره می شود یعنی اولین بایت میزان سفیدی آخرین پیکسل تصویر است و آخرین بایت، میزان سفیدی اولین پیکسل تصویر است. رنگ هر پیکسل عددی بین ۰ تا ۲۵۵ است. می‌توان با تخصیص میزان سفیدی هر پیکسل به مختصات z نقاط، ناهمواریهایی منطبق بر الگو ایجاد کرد.

برای ایجاد زمینی دلخواه و زیبا، به جای اینکه آرایه Points را بصورت دستی پر کنیم، خانه های آن را از روی یک فایل پر خواهیم کرد. برای این منظور ابتدا یک تصویر ۶۴*۶۴ پیکسلی سیاه و سفید را بارگذاری کرده، و از مقدار سفیدی هر پیکسل به عنوان مقدار z نقطه متناظر آن در زمین، استفاده خواهیم کرد.

ابتدا خط زیر را به پروژه اضافه می نمائیم:

```
using System.IO;
```

مختصات زمین را نیز ۶۴*۶۴ تنظیم می نمائیم:

```
private int Height = 64;
private int Width = 64;
```

سپس متد FillPoints را به شکل زیر تغییر می دهیم:

```
private void FillPoints()
{
  Points = new int[Width, Height];
  FileStream fs = new FileStream("abc.raw", FileMode.Open,
                                 FileAccess.Read);
  BinaryReader r = new BinaryReader(fs);
  for (int i = 0; i < Height; i++)
  {
    for (int y = 0; y < Width; y++)
    {
      int H = (int)(r.ReadByte() / 50);
      Points[Width - 1 - y, Height - 1 - i] = H;
```

```
    }
  }
  r.Close();
}
```

ابتدا آرایه ای ۶۴*۶۴ جهت نگهداری مختصات z نقاط تعریف شده است. خط بعدی یک فایل تصویری به نام abc.raw (که باید در کنار فایل اجرائی برنامه باشد) را باز می نماید. در یک فایل raw میزان سفیدی هر پیکسل به صورت بایت به بایت در فایل نوشته می شود، بنابراین تنها کاری که باید انجام دهیم بارگذاری مقدار هر بایت خوانده شده از فایل در خانه بعدی آرایه است. در خط دوم یک BineryReader جهت دسترسی به اطلاعات فایل تعریف شده است. از آنجایی که مقدار سفیدی هر پیکسل عددی بین ۰ تا ۲۵۵ است، این مقدار بر ۵۰ تقسیم شده تا مختصات z نقاط زیاد بزرگ نباشد. همچنین چون اطلاعات پیکسلها در فایل raw به صورت برعکس (از انتهای فایل) نوشته شده است از width-1 برای معکوس کردن جایگذاری استفاده شده است. یعنی آخرین بایت خوانده شده در اولین خانه آرایه قرار خواهد گرفت.

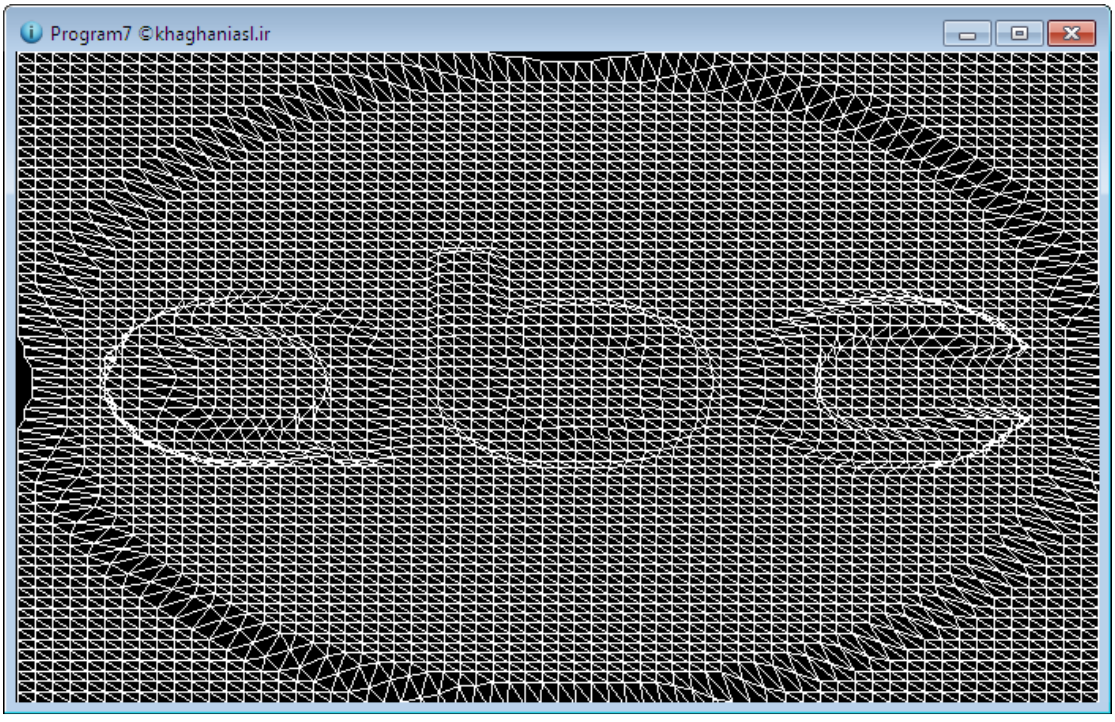
قبل از رسم مثلثها، آنها را به گونه ای انتقال می دهیم که وسط زمین در نقطه (۰،۰،۰) قرار گیرد. برای این منظور دستور زیر را قبل از فراخوانی DrawIndexedPrimitives در متد OnPaint قرار می دهیم:

```
device.Transform.World= Matrix.Translation(-Height/2,-Width/2,0);
```

پس از اینکه زمین به مرکز مختصات منتقل شد، تنها کار باقی مانده، تغییر موقعیت دوربین است. پس متد Camera را دستخوش تغییرات زیر قرار می دهیم:

```
private void Camera()
{
  device.Transform.Projection = Matrix.PerspectiveFovLH(
        (float)Math.PI/4, this.Width/this.Height, 1f, 150f);
  device.Transform.View = Matrix.LookAtLH(new Vector3(0, 0, 80),
                    new Vector3(0, 0, 0), new Vector3(0, 1, 0));
  device.RenderState.Lighting = false;
}
```

توجه داشته باشید که ClipingPlane از ۵۰ به ۱۵۰ تغییر یافته تا کل زمین قابل مشاهده باشد. اگر رنگ پس زمینه را در متد OnPaint به سیاه تغییر دهید، خروجی زیباتری خواهید دید:



۲-۵-۳ ایجاد زمین از روی فایل bmp

از آنجایی که فایلهای bmp یکی از گسترده ترین فرمتهای ذخیره سازی تصاویر بوده و توسط اغلب نرم افزارها قابل ویرایش و استفاده هستند، در این بخش نحوه استفاده از فایلهای bmp برای شکل دهی زمین بررسی می شود.

دلیل استفاده از فایلهای raw برای شروع کار این بود که تمام بایتهای خوانده شده از فایل raw قابل استفاده هستند. در یک فایل bmp علاوه بر رنگ پیکسلها، اطلاعات دیگری از قبیل مقدار طول و عرض تصویر، اندازه فایل و... نیز نگهداری می شود. این موضوع استفاده از فایلهای bmp را اندکی دشوارتر از فایلهای raw می کند زیرا اطلاعات اضافی نباید استفاده شوند.

در یک فایل bmp برعکس فایل raw برای هر پیکسل ۳ بایت در نظر گرفته شده است که هر یک از بایتها میزان یکی از سه رنگ استفاده شده در آن پیکسل را مشخص می کند. ۵۳ بایت اول فایل bmp هدری است که درباره خود فایل است. بایت ۱۱ تا ۱۴ آفست شروع اطلاعات می باشد. بایت ۱۹ تا ۲۲ عرض تصویر و ۲۳ تا ۲۶ ارتفاع تصویر است.

در ابتدای هر فایل bmp و قبل از اینکه اطلاعات مربوط به پیکسلهای تصویر نوشته شود، هدری که اطلاعاتی در مورد تصویر را نگهداری می کند، قرار می گیرد. از آنجایی که فقط کدهای ۰ تا ۲۵۵ می توانند توسط ویرایشگرها نشان داده شود، باز نمودن یک فایل bmp توسط یک ویرایشگر، اطلاعات مفیدی در مورد این هدر نخواهد داد. در جدول زیر ساختار هدر فایل bmp نشان داده شده است:

Byte	Nr	Value
1-2	66 77	Always BM
3-6	X X X X	File Size
7-10	0 0 0 0	Always 0
11-14	X X X X	Offset to Pixel Data
15-18	40 0 0 0	Always 40
19-22	X X X X	Image width in Pixels
23-26	X X X X	Image height in Pixels
27-28	1 0	Always 1
29-32	24 0 0 0	Bits per Pixel
33-36	0 0 0 0	Compression, usually 0
37-53	4x(0 0 0 0)	No longer used

همانطور که ملاحظه می شود دو بایت اول هر فایل bmp دارای مقادیر ۶۶ و ۷۷ می باشد که کد اسکی حروف BM می باشند. مهمترین اطلاعات موجود در این جدول، طول و عرض تصویر و آفست محلی است که اطلاعات پیکسلها از آنجا آغاز می شود. از آنجایی که طول هدر همیشه ۵۳ بایت است، بنابراین آفست شروع اطلاعات پیکسلها نیز همیشه ۵۴ است. همانطور که ملاحظه می شود تعداد بیتهای استفاده شده برای نگهداری اطلاعات هر پیکسل نیز ۲۴ بیت یا ۳ بایت می باشد. هر بایت می تواند مقداری بین ۰ تا ۲۵۵ را داشته و مقدار یکی از ۳ رنگ اصلی در آن پیکسل را نشان دهد. یک بایت برای قرمز، یک بایت برای سبز و یک بایت برای رنگ آبی است.

برای استفاده از اطلاعات پیکسلها به عنوان مختصات z نقاط زمین، ابتدا باید آفست شروع اطلاعات پیکسلها را یافته و مقدار ۳ بایت متناظر با هر پیکسل را جمع کنیم تا میزان سفیدی آن پیکسل و در نتیجه مختصات z نقطه متناظر در زمین معلوم شود.

در ابتدای متد FillPoints دو متغیر به شکل زیر تعریف می کنیم:

```
int offset;
byte dummy;
```

از متغیر offset برای نگهداری آفست و از متغیر dummy جهت نگهداری بایتهای خوانده شده از فایل که لازم نیستند، استفاده می کنیم. همانند بخش قبل فایل را باز کرده و یک شی BinaryReader را به آن وصل می کنیم:

```
FileStream fs = new FileStream("Picture.bmp", FileMode.Open,
                               FileAccess.Read);
BinaryReader r = new BinaryReader(fs);
```

برای به دست آوردن آفست اطلاعات پیکسلها، ۱۰ بایت اول هدر باید رد شود. بنابراین دستور بعدی را به شکل زیر می نویسیم:

```
for (int i = 0; i < 10; i++)
{
  dummy = r.ReadByte();
}
```

حال ۴ بایت بعدی که خوانده خواهند شد، آفست اطلاعات پیکسلها را نشان خواهد داد. از آنجایی که هر بایت حداکثر می تواند عدد ۲۵۵ را در خود نگه دارد، اولین بایت خوانده شده در ۱، دومی در ۲۵۵، سومی در ۲۵۵*۲۵۵ و چهارمی در ۲۵۵*۲۵۵*۲۵۵ ضرب می شود:

```
offset = r.ReadByte();
offset += r.ReadByte() * 256;
offset += r.ReadByte() * 256 * 256;
offset += r.ReadByte() * 256 * 256 * 256;
```

با توجه به جدول، برای خواندن اطلاعات طول و عرض تصویر ۴ بایت دیگر باید رد شود. بنابراین در ادامه خواهیم داشت:

```
for (int i = 0; i < 4; i++)
{
  dummy = r.ReadByte();
}
Width = r.ReadByte();
Width += r.ReadByte() * 256;
Width += r.ReadByte() * 256 * 256;
Width += r.ReadByte() * 256 * 256 * 256;
Height = r.ReadByte();
Height += r.ReadByte() * 256;
Height += r.ReadByte() * 256 * 256;
Height += r.ReadByte() * 256 * 256 * 256;
```

حال می توان آرایه Points را ایجاد کرده و به آفستی که اطلاعات پیکسلها از آنجا آغاز می شود، منتقل شد. از آنجایی که ۲۶ بایت تاکنون از فایل خوانده ایم، بنابراین کافیست offset - 26 بایت باقیمانده را جهت رسیدن به اطلاعات اولین پیکسل بخوانیم:

```
Points = new int[Width, Height];
for (int i = 0; i < (offset - 26); i++)
{
  dummy = r.ReadByte();
}
```

تنها کار باقیمانده خواندن اطلاعات پیکسلها و قرار دادن آنها در آرایه می باشد. پس برای تکمیل متد FillPoints کدهای زیر را در ادامه قرار می دهیم:

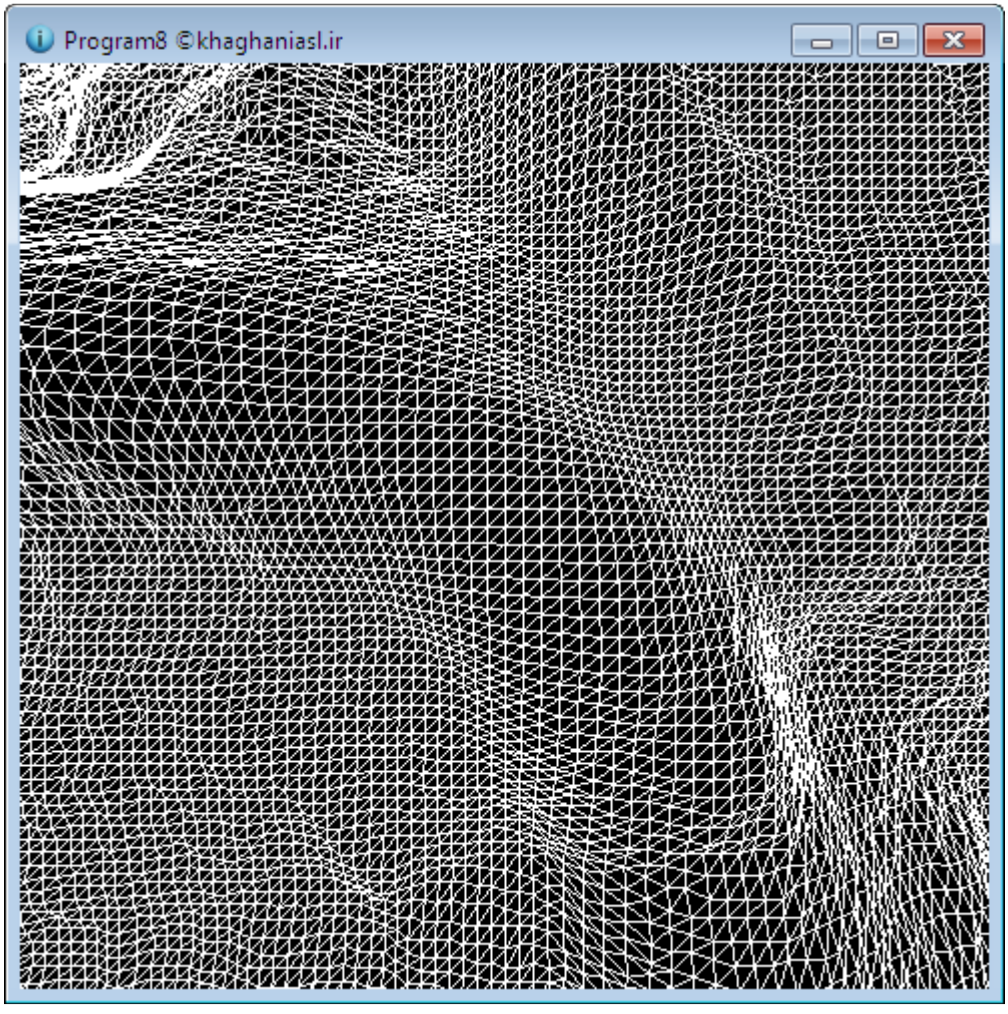
```
for (int i = 0; i < Height; i++)
{
  for (int y = 0; y < Width; y++)
  {
    int H = (int)(r.ReadByte());
    H += (int)(r.ReadByte());
```

```
            H += (int)(r.ReadByte());
            H /= 8;
            Points[Width - 1 - y, Height - 1 - i] = H;
        }
    }
    r.Close();
```

از آنجایی که عکس مورد استفاده ۱۲۸*۱۲۸ پیکسلی است، در متد Camera موقعیت دوربین را نیز کمی تغییر داده و Clipping Plane را به ۱۵۰ افزایش می دهیم:

```
device.Transform.Projection = Matrix.PerspectiveFovLH(
            (float)Math.PI/4, this.Width/this.Height, 1f, 150f);
device.Transform.View = Matrix.LookAtLH(new Vector3(0, 0, -50),
new Vector3(0, 0, 0), new Vector3(0, 1, 0));
```

حال با اجرای برنامه، نقاط سیاه و سفید موجود در عکس را به شکل پستی و بلندیهایی مشاهده خواهید کرد:



۲-۶ حرکت در فضا

برای حرکت بایستی مختصات دوربین توسط صفحه کلید یا ماوس تغییر داده شود. یکی از زیرمجموعه های مهم DirectX، DirectInput می باشد که شامل کلاسهایی برای کنترل صفحه کلید، ماوس و دسته های بازی است. DirectInput نیز در داخل خود device دارد که سخت افزار مربوط به صفحه کلید، ماوس یا دسته های بازی را با آن کنترل می کند.

۲-۶-۱ چرخش زمین

استفاده از صفحه کلید در DirectX بسیار ساده است. ابتدا namespace زیر را به رفرنسهای پروژه اضافه می کنیم:

```
using Microsoft.DirectX.DirectInput;
```

توجه: نوع Device در دو فضای نام Microsoft.DirectX.Direct3D و Microsoft.DirectX.DirectInput با کاربردهایی متفاوت وجود دارد و برای تمایز بین نوع Device در این دو فضای نام، کافی است در دو خط از برنامه قبلی اعلان کنیم که نوع Device موجود، از فضای نام Microsoft.DirectX.Direct3D می باشد:

```
private Microsoft.DirectX.Direct3D.Device device;
device = new Microsoft.DirectX.Direct3D.Device(0,
Microsoft.DirectX.Direct3D.DeviceType.Reference, this,
CreateFlags.SoftwareVertexProcessing , pp);
```

ابتدا دو متغیر زیر را به کلاس اضافه می نمائیم:

```
private Microsoft.DirectX.DirectInput.Device keyboard;
private float angle = 0f;
```

متغیر keyboard جهت ارتباط با صفحه کلید و متغیر angle بمنظور تعیین زاویه چرخش زمین است.

متدی به نام InitializeKeyboard به شکل زیر جهت پیکربندی صفحه کلید تعریف می نمائیم:

```
public void InitializeKeyboard()
{
  keyboard = new Microsoft.DirectX.DirectInput.Device
            (SystemGuid.Keyboard);
  keyboard.SetCooperativeLevel(this,CooperativeLevelFlags.Background|
                               CooperativeLevelFlags.NonExclusive);
  keyboard.Acquire();
}
```

خط اول، صفحه کلید پیش فرض سیستم را به متغیر keyboard انتساب می دهد. در خط دوم، برخی از رفتارهای صفحه کلید در متغیر keyboard و نحوه به اشتراک گذاری صفحه کلید با سایر برنامه های در حال اجرا تعیین شده است. در نهایت با فراخوانی متد Acquire شی ایجاد شده عملا با سخت افزار درگیر شده و داده های ارسالی از آن را دریافت خواهد کرد. لازم به یادآوری است که متد فوق بایستی در تابع سازنده فرم فراخوانی شود.

```
public Form1()
{
  …
  InitializeKeyboard();
}
```

متدی به نام ReadKeyboard تعریف می کنیم تا به ازای فشردن و نگهداشتن کلیدهای جهتی چپ و راست، به متغیر angle (زاویه چرخش) 0.5f اضافه یا کاسته شود. این متد بایستی در متد OnPaint فراخوانی گردد:

```
private void ReadKeyboard()
{
  KeyboardState keys = keyboard.GetCurrentKeyboardState();
  if (keys[Key.RightArrow])
      angle -= 0.5f;
  if (keys[Key.LeftArrow])
      angle += 0.5f;
}
```

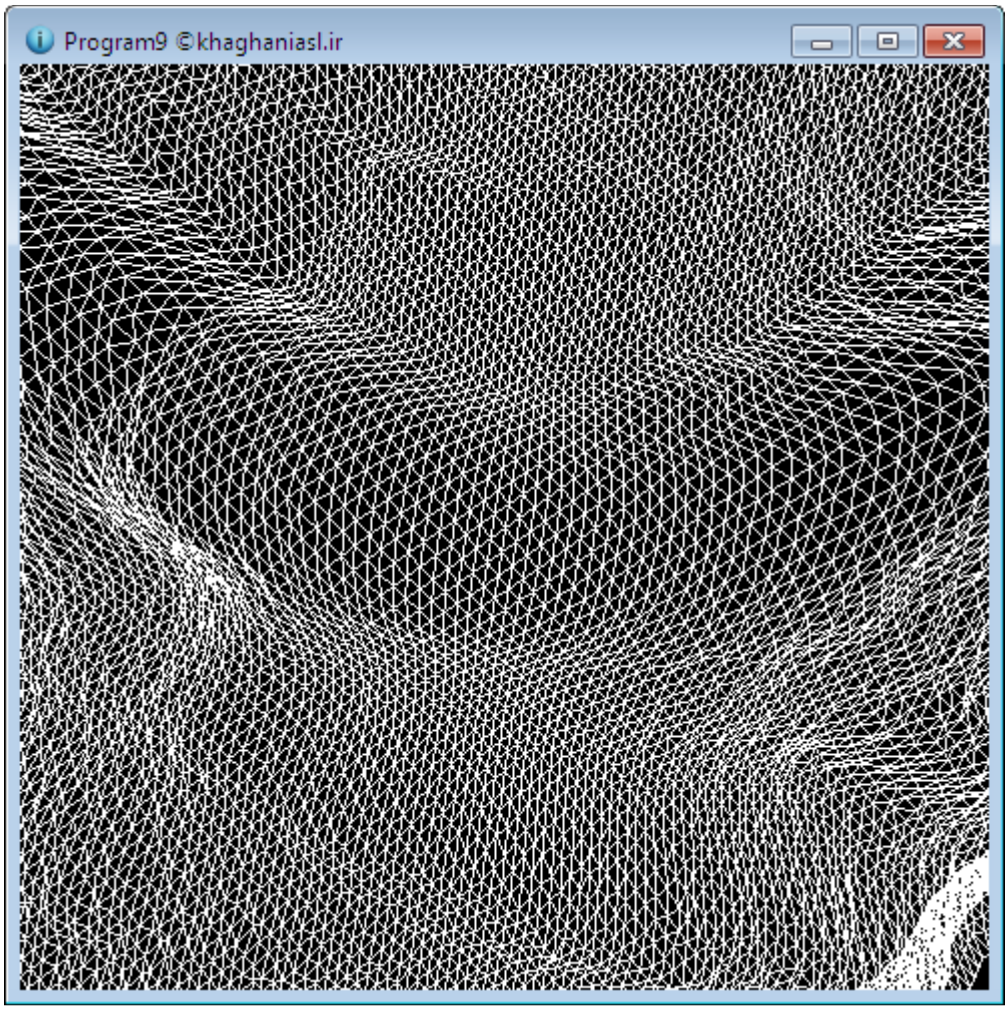
برای دریافت داده های صفحه کلید، متغیری از نوع KeyboardState تعریف کرده ایم. متد GetCurrentKeyboardState آخرین وضعیت کلیدهای فشرده شده را برمی گرداند. برای بررسی فشرده شدن

کلیدی در صفحه کلید، می توان کلید مربوطه را به صورت اندیس در KeyboardState استفاده کرد. به عنوان مثال جهت بررسی فشرده شدن کلید RightArrow دستوری همانند خط دوم را می توان نوشت.

حال کافی است متد OnPaint را دستخوش تغییرات زیر قرار دهیم تا با فشردن و نگهداشتن کلیدهای جهتی چپ و راست، مقدار متغیر angle تغییر کرده و زمین طراحی شده بصورت ساعتگرد یا پادساعتگرد بچرخد.

```
protected override void OnPaint(PaintEventArgs e)
{
  device.Clear(ClearFlags.Target,Color.Black, 0, 1);
  device.BeginScene();
  device.VertexFormat = CustomVertex.PositionColored.Format;
  device.SetStreamSource(0, vb, 0);
  device.Indices = ib;
  device.Transform.World = Matrix.Translation(-Height/2,-Width/2,
                           0) * Matrix.RotationZ(angle);
  device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                         Width * Height, 0, indices.Length / 3);
  device.EndScene();
  device.Present();
  ReadKeyboard();
  this.SetStyle(ControlStyles.AllPaintingInWmPaint |
                ControlStyles.Opaque, true);
  this.Invalidate();
}
```

با اجرای برنامه و فشردن کلیدهای جهتی راست یا چپ، زمین بصورت ساعتگرد یا پادساعتگرد می چرخد:



برای استفاده از ماوس نیز دقیقا همانند صفحه کلید، کافی است شی ای از نوع device ایجاد کرده و آن را با ماوس مرتبط کنیم. ابتدا متغیر زیر را به کلاس اضافه می نمائیم:

```
private Microsoft.DirectX.DirectInput.Device Mouse;
```

سپس متدی به نام InitializeMouse به شکل زیر جهت پیکربندی ماوس تعریف می نمائیم:

```
public void InitializeMouse()
{
  Mouse = new Microsoft.DirectX.DirectInput.Device
            (SystemGuid.Mouse);
  Mouse.SetCooperativeLevel(this,CooperativeLevelFlags.Background|
                             CooperativeLevelFlags.NonExclusive);
  Mouse.Acquire();
}
```

لازم به یادآوری است که متد فوق بایستی در تابع سازنده فرم فراخوانی شود.

```
public Form1()
{
  …
  InitializeMouse();
}
```

متدی به نام ReadMouse تعریف می کنیم تا به ازای فشردن و نگهداشتن دکمه های چپ و راست ماوس، به متغیر angle (زاویه چرخش) 0.5f اضافه یا کاسته شود. این متد بایستی در متد OnPaint فراخوانی گردد.

```
private void ReadMouse()
{
   MouseState MState = Mouse.CurrentMouseState;
   if (MState.GetMouseButtons()[0]>0)
     angle -= 0.5f;
   if (MState.GetMouseButtons()[1] > 0)
     angle += 0.5f;
}
```

برای دریافت اطلاعات ماوس از کلاسی به نام MouseState استفاده می شود. وضعیت جاری ماوس در شی MState برگشت داده می شود. منظور از وضعیت جاری ماوس؛ مکان ماوس و وضعیت دکمه های ماوس است. شی MState دارای سه خصوصیت x و y و z می باشد که مسافت پیموده شده در راستای مربوطه از آخرین محل توقف ماوس را نشان می دهد. مثلا هنگام حرکت ماوس در راستای محور X، به اندازه حرکت داده شده چند میلی ثانیه x تغییر می کند. لازم به ذکر است حرکت ماوس تنها روی x و y تاثیر داشته و z را غلتک ماوس تغییر می دهد. متد GetMouseButtons در این شی وضعیت کلیدهای فشرده شده را به صورت آرایه ای از بایت بر می‌گرداند. بایت 0 معرف دکمه چپ ماوس، بایت 1 معرف دکمه راست ماوس و بایت 2 معرف دکمه وسط (غلتک ماوس) است.

حال کافی است متد ReadMouse را داخل متد OnPaint فراخوانی کنیم تا با فشردن دکمه های چپ و راست ماوس، مقدار متغیر angle تغییر کرده و زمین طراحی شده بصورت ساعتگرد یا پادساعتگرد بچرخد.

```
protected override void OnPaint(PaintEventArgs e)
{
  …
  ReadKeyboard();
  ReadMouse();
  …
}
```

از آنجایی که صفحه کلید و ماوس نیز همانند کارت گرافیکی بصورت یک device در DirectX شناخته می‌شود، بهتر است کدهای مربوط به کنترل آنها را در کلاس دیگری قرار دهیم.

۲-۷ استفاده از رنگ

بدیهی است اگر سطح زمین، به جای خطوط با رنگهائی پوشیده شود بهتر دیده خواهد شد. به عنوان مثال مطلوب است از رنگهایی طبیعی استفاده کنیم مثلا دره ها با رنگ آبی پر شوند تا نشان دهنده دریاچه ها باشد، رنگ سبز برای نشان دادن جنگل و رنگ قهوه ای برای نشان دادن کوهها و در نهایت رنگ سفید جهت نشان دادن برف قله ها استفاده شود.

نمی‌توان انتظار داشت که دریاچه ها همیشه در ارتفاع صفر و قله ها همیشه در ارتفاع ۲۵۵ باشد زیرا در این صورت تصویری که مقادیر تمام پیکسلهای آن بین ۵۰ تا ۲۰۰ است هیچ دریاچه یا قله ای تولید نخواهد کرد. بنابراین اولین کاری که انجام می دهیم به دست آوردن حداقل و حداکثر ارتفاع تصویری است که برای زمین انتخاب شده است. بنابراین دو متغیر به صورت زیر در کلاس تعریف می کنیم:

```
private int MinHeight = 0;
private int MaxHeight = 255;
```

هنگام بارگذاری پیکسلهای تصویر، با استفاده از تکه کد زیر که به متد FillPoints افزوده شده، می توان حداکثر و حداقل ارتفاع تصویر را مشخص کرد:

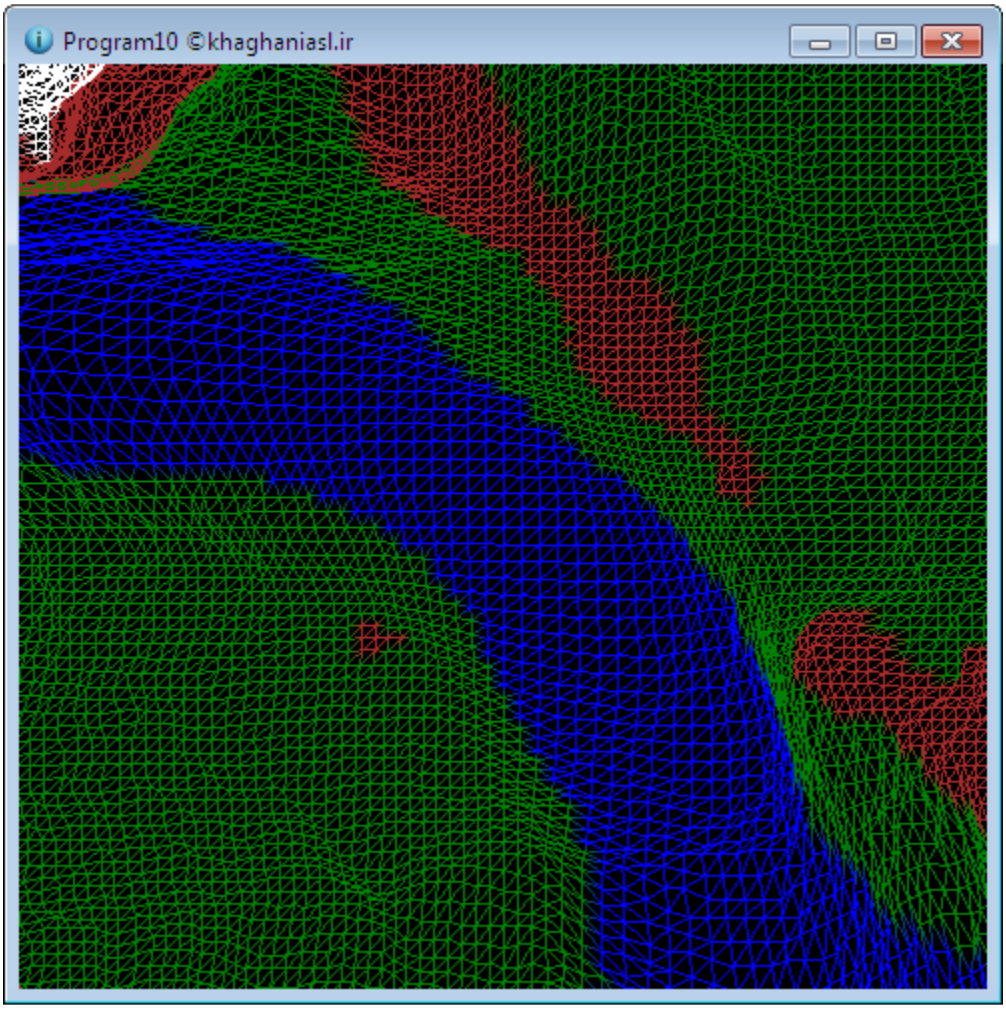
```
private void FillPoints()
{
  ...
  ...
  for (int i = 0; i < Height; i++)
  {
    for (int y = 0; y < Width; y++)
    {
      int H = (int)(r.ReadByte());
      H += (int)(r.ReadByte());
      H += (int)(r.ReadByte());
      H /= 8;
      Points[Width - 1 - y, Height - 1 - i] = H;
      if (H < MinHeight)
        MinHeight = H;
      if (H > MaxHeight)
        MaxHeight = H;
    }
  }
  r.Close();
}
```

حال به سادگی می توان هنگام تعیین مختصات و رنگ مثلثها، با تغییراتی در متد VertexDeclaration، با توجه به ارتفاع نقطه، رنگ آن را مشخص کرد:

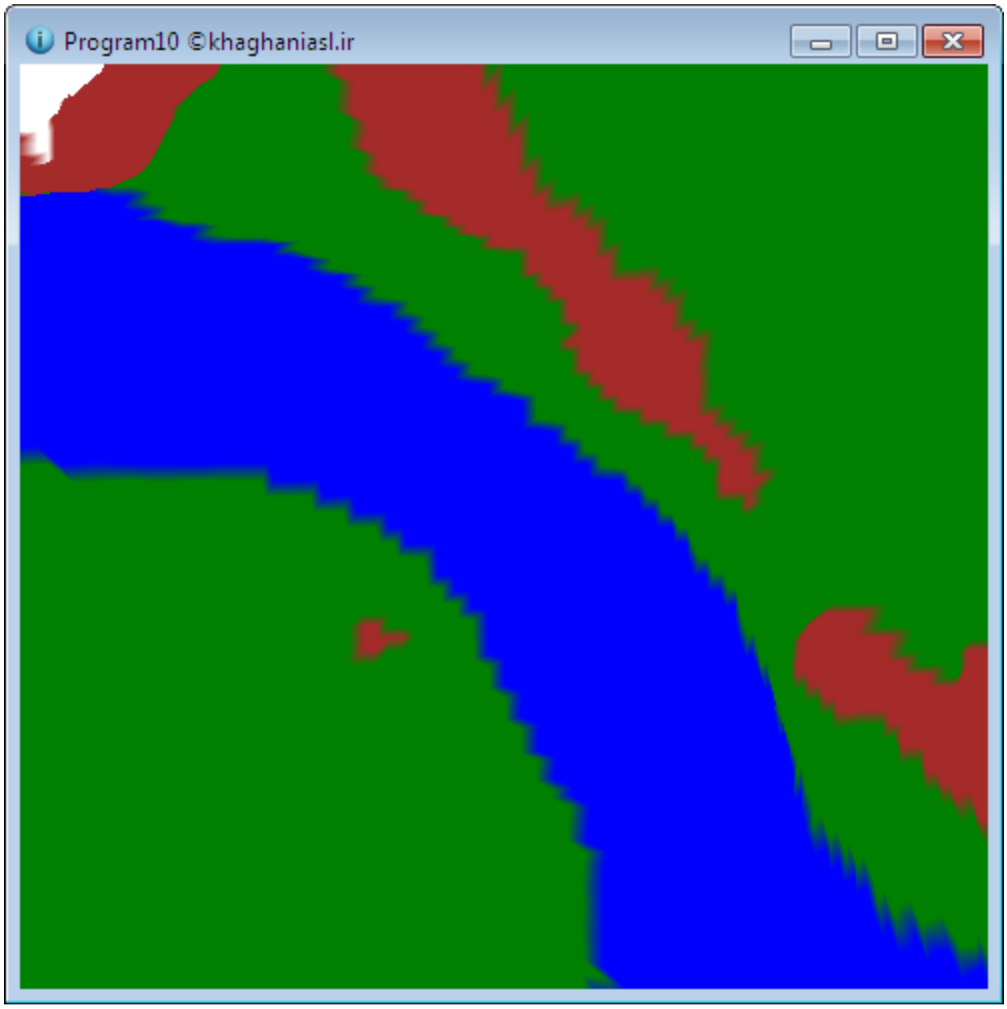
```
private void VertexDeclaration()
{
  vb = new VertexBuffer(typeof(CustomVertex.PositionColored),
         Width * Height, device, Usage.Dynamic | Usage.WriteOnly,
         CustomVertex.PositionColored.Format, Pool.Default);
  vert = new CustomVertex.PositionColored[Width * Height];
  for (int x = 0; x < Width; x++)
  {
    for (int y = 0; y < Height; y++)
    {
      vert[x+ y * Width].Position = new Vector3(x,y,Points[x,y]);
      if (Points[x,y]< MinHeight + (MaxHeight - MinHeight)/4)
        vert[x + y * Width].Color = Color.Blue.ToArgb();
```

```
        else if (Points[x,y]< MinHeight+(MaxHeight-MinHeight)*2/4)
          vert[x + y * Width].Color = Color.Green.ToArgb();
        else if (Points[x,y]< MinHeight +(MaxHeight-MinHeight)*3/4)
          vert[x + y * Width].Color = Color.Brown.ToArgb();
        else
          vert[x + y * Width].Color = Color.White.ToArgb();
      }
    }
    vb.SetData(vert, 0, LockFlags.None);
}
```

حال با اجرای برنامه، تصویری مشابه شکل زیر مشاهده خواهید کرد:



می توان با حذف خط device.RenderState.FillMode = FillMode.WireFrame در متد InitializeDevice زمینی کاملا رنگی ایجاد کرد:



از لحاظ رنگ آمیزی چون فقط x و y در نظر گرفته شده و z را در نظر نگرفته ایم، گاهی رنگها در خروجی با هم تداخل پیدا می کنند. اگر برنامه را اجرا کرده و آن را دوران دهید، خواهید دید که گاهی مرکز زمین توسط دریاچه پشت آن برش داده می شود. دلیل این امر، استفاده نکردن از ZBuffer است.

ZBuffer آرایه ای است که به کمک آن، پردازنده کارت گرافیکی می تواند مختصات عمقی تمامی پیکسلهای در حال نمایش را نگهداری کند. ZBuffer به ازای هر پیکسل عددی را نگه می‌دارد. هنگام رسم هر مثلث توسط کارت گرافیکی، مختصات مثلث جدید با پیکسلهای قبلی رسم شده در آن محل که در ZBuffer نگهداری می شوند، مقایسه می شود تا مشخص شود که مثلث جدید از لحاظ ارتفاع به صفحه نمایش نزدیکتر است یا خیر. اگر مثلث جدید نزدیکتر باشد، محتویات ZBuffer در آن محل به روز رسانی می شود.

Front Buffer بافری است که محتویات آن در هر لحظه ۶۰ بار در صفحه نمایش نشان داده می شود. (اگر فرکانس صفحه نمایش ۶۰ هرتز باشد) به عبارت دیگر رنگهایی را نشان می دهد که همان لحظه در صفحه نمایش در حال نمایش است. Back Buffer نیز بافری است که هر کدی که برنامه نویس بنویسد در آن اعمال می شود. با فراخوانی متد Present محتویات آن در Front Buffer کپی شده و اشیای ساخته شده، نمایش داده می شود. اما هنگام کپی، ZBuffer بررسی می شود. اگر عمق Back Buffer از آن بیشتر باشد، پیکسل موجود عوض نمی شود یعنی تنها پیکسلهایی که به دوربین نزدیکترند رسم خواهند شد.

البته تمام این کارها به صورت اتوماتیک انجام می شود و تنها کاری که برنامه نویس باید انجام دهد، تعریف ZBuffer هنگام ایجاد device است. بنابراین دو خط زیر به انتهای دستورات پیکربندی PresentParameters در متد InitializeDevice اضافه می کنیم:

```
pp.AutoDepthStencilFormat = DepthFormat.D16;
pp.EnableAutoDepthStencil = true;
```

در خط اول ZBufferای با دقت ۱۶ بیت ایجاد شده است. یعنی به ازای هر پیکسل یک عدد دو بایتی در نظر گرفته می شود. ارتفاع تصویر به صورت رشته ای بیتی از پیکسلها در نظر گرفته می شود که فاصله دو نقطه متوالی از یکدیگر ۱ است. با ۱۶ بیت حداکثر می توان 2^{16} یعنی ۶۵۵۳۶ را نگهداری کرد. در نتیجه اگر ارتفاع تصویر، بزرگتر از این مقدار باشد ۱۶ بیت کافی نخواهد بود. البته بیشتر کارتهای گرافیکی از ۳۲ بیت نیز برای ZBuffer پشتیبانی می کنند اما ۱۶بیت نیز برای مثالهای ما بسیار زیاد است. خط دوم ZBuffer تعریف شده را فعال کرده و منجر به کنترل عمق و سایه می شود.

اگر برنامه را همین لحظه اجرا کنید، صحنه‌ای تقریبا سیاه را مشاهده خواهید کرد. دلیل این اتفاق عدم مقداردهی اولیه به ZBuffer است. از آنجایی که ZBuffer ابتدا با صفر مقداردهی می شود بنابراین تمام مثلثهای رسم شده دورتر از پیکسلهای فعلی موجود در ZBuffer نگاشته شده و حذف می شوند. بنابراین باید قبل از رسم زمین تمام بیتهای موجود در ZBuffer را با یک مقداردهی کنیم. برای این کار کافی است در متد OnPaint، خط مربوط به تابع Clear را به شکل زیر تغییر دهیم:

```
device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.Black,
             1, 0);
```

پارامتر سوم بزرگترین عدد ممکن را برای ZBuffer در نظر می گیرد و همه zها از این مقدار کوچکتر شده و رسم می شوند. پارامتر چهارم نیز مربوط به سایه زنی است. هر چند با استفاده از ZBuffer تمامی مشکلات از جمله مشکل تداخل رنگ برطرف خواهد شد ولی با این حال، هنوز زمین طبیعی به نظر نمی رسد. در بخش بعد نحوه استفاده از نور و سایه ها برای طبیعی جلوه دادن محیط شرح داده شده است.

۲-۸ استفاده از نور

با وجود استفاده از رنگها و ZBuffer، زمین طراحی شده در بخش قبل هنوز هم طبیعی به نظر نمی رسد. با اضافه نمودن مقداری نور، محیط بسیار بهتر دیده خواهد شد. در DirectX سه نوع منبع نور وجود دارد:

۱- Point: این نوع منبع نوری نور را در تمام جهات پخش می کند مانند یک لامپ.
۲- Directional: این منبع نوری نور را در یک جهت ثابت و مستقیم پخش می کند مانند خورشید.
۳- Spot: این منبع نوری نور را در یک جهت و در یک شعاع محدود پخش می کند مانند نوری که در سن پخش می شود.

اگر بخش دوران و انتقال را به خاطر داشته باشید، آنجا جهت مشاهده مثلثها وضعیت نور پردازی در محیط را خاموش کردیم. حال دوباره به همان پروژه برگشته و دستور مربوط به نحوه نورپردازی محیط را در متد OnPaint تصحیح می کنیم.

```
device.RenderState.Lighting = true;
```

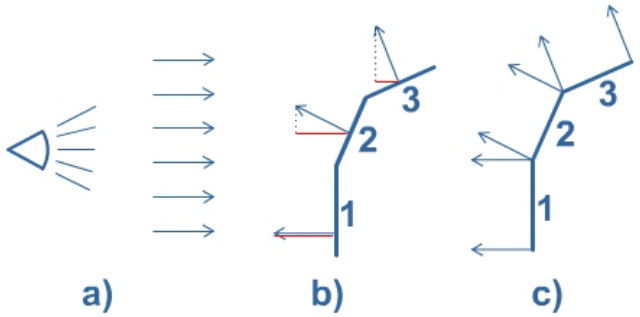
بلافاصله پس از این دستور، می توان چراغهای لازم برای نورپردازی محیط را تعریف کرد:

```
device.Lights[0].Type = LightType.Directional;
device.Lights[0].Diffuse = Color.White;
device.Lights[0].Direction = new Vector3(0.8f, 0, -1);
device.Lights[0].Enabled = true;
```

در تکه کد فوق از ساده ترین نوع نورها در DirectX یعنی نور مستقیم استفاده شده است. در این نوع نور، پرتوهای نوری در مسیری مستقیم حرکت می کنند. خصوصیت Diffuse، رنگ نور را مشخص می کند. همچنین مسیر نورافشانی چراغ نیز باید مشخص شود. در انواع دیگر منابع نوری، می توان برخی دیگر از خصوصیات همچون

Attenuation برای ضریب فرسایش نور، Range شعاعی که منبع نور در آن فعال است و Position برای تنظیم محل منبع نور را نیز استفاده کرد. ممکن است برخی از کارتهای گرافیکی، نورپردازی در محیط را پشتیبانی نکنند.

اگر برنامه را این لحظه، اجرا کنید همچنان صفحه ای تاریک را مشاهده خواهید کرد. جهت اعمال نور به محیط DirectX به پیکربندی پارامتر دیگری نیز نیاز داریم. به شکلهای زیر توجه کنید:



اگر منبع نوری به شکل a داشته و آن را به سطح شی ای مشابه b بتابانیم چگونه DirectX متوجه خواهد شد که سطح ۱ بسیار درخشانتر از سطح ۲ باید دیده شود؟ اگر به خطوط قرمز رسم شده توجه کنید نحوه انعکاس طبیعی نور پس از برخورد به سطح شی را مشاهده خواهید کرد. حال چگونه باید مسیر انعکاس نورها پس از برخورد به یک سطح را محاسبه کرد؟

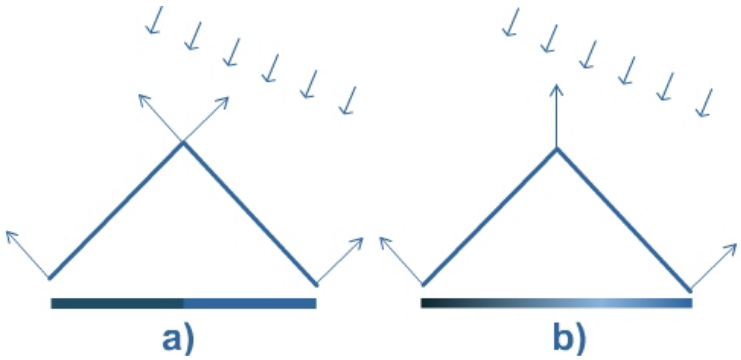
انجام این محاسبات دشوار را خود DirectX بر عهده گرفته است و تنها کاری که برنامه نویس باید انجام دهد تعریف مسیر خطوط برگشتی آبی رنگ با زاویه ۹۰ درجه نسبت به هر یک از سطوح است. بقیه کارها توسط DirectX انجام خواهد شد. از آنجایی که می دانیم هر سطح از ترکیب مثلثها ساخته می شود، می توانیم این خطوط برگشتی را هنگام تعریف نقاط مثلثها تعریف کنیم. این کار با تغییر نوع نقاط به CustomVertex.PositionNormalColored انجام می شود:

```
CustomVertex.PositionNormalColored[] vert;
```

بدیهی است این تغییر را بایستی به اطلاع Device نیز برسانیم:

```
device.VertexFormat = CustomVertex.PositionNormalColored.Format;
```

حال باید خطوط برگشتی را برای هر سطح تعیین کنیم اما قبل از این کار، نگاهی به شکل زیر بیندازید:



در شکل فوق، خطوطی که از سمت بالا کشیده شده اند؛ جهت تابش نور، و خط رنگی ضخیم پائین شکل b رنگ مورد انتظار پیکسلها در سطح مثلث را نشان می دهد. اگر صرفا خط برگشتی هر سطح را مشخص کنیم، یک برش رنگی بر روی سطح مثلث به وجود آمده (شکل a) و شکل به دو مثلث مجزا تبدیل خواهد شد. جهت حل این مشکل اگر راس بالائی مثلث را به عنوان یک راس normal تعریف کنیم DirectX ترتیب بقیه کارها را داده و شکلی بسیار نرمتر مانند شکل b را ایجاد خواهد کرد. بدیهی است مختصات عرضی این راس، نصف مجموع دو راس کناری می-باشد. برای پیاده سازی این امر، موقعیت دوربین را به شکل زیر تغییر می دهیم:

```
device.Transform.Projection = Matrix.PerspectiveFovLH(
            (float)Math.PI/4, this.Width/this.Height, 1f, 200f);
device.Transform.View = Matrix.LookAtLH(new Vector3(0, -40, 100),
                    new Vector3(0, 50, 0), new Vector3(0, 1, 0));
```

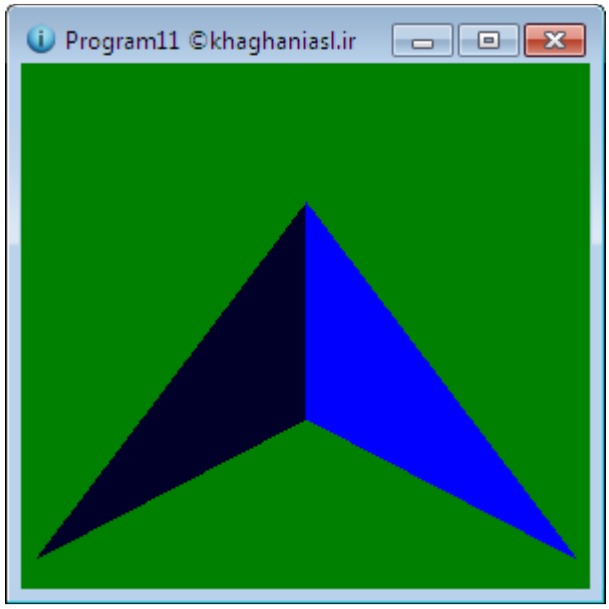
موقعیت چراغ همانند مثال قبلی است و مقدار مثبت برای x به معنی حرکت به سمت چپ و مقدار منفی برای z به معنی حرکت به سمت پائین است. بنابراین متد DrawTriangle را به صورت زیر تغییر می دهیم:

```
public void DrawTriangle()
{
  vert = new CustomVertex.PositionNormalColored[6];
  vert[0].Position = new Vector3(0f, 0f, 40f);
  vert[0].Color = Color.Blue.ToArgb();
  vert[0].Normal = new Vector3(1, 0, 1);
  vert[1].Position = new Vector3(40f, 0f, 0f);
  vert[1].Color = Color.Blue.ToArgb();
  vert[1].Normal = new Vector3(1, 0, 1);
  vert[2].Position = new Vector3(0f, 40f, 40f);
  vert[2].Color = Color.Blue.ToArgb();
  vert[2].Normal = new Vector3(1, 0, 1);
  vert[3].Position = new Vector3(-40f, 0f, 0f);
  vert[3].Color = Color.Blue.ToArgb();
  vert[3].Normal = new Vector3(-1, 0, 1);
  vert[4].Position = new Vector3(0f, 0f, 40f);
  vert[4].Color = Color.Blue.ToArgb();
  vert[4].Normal = new Vector3(-1, 0, 1);
  vert[5].Position = new Vector3(0f, 40f, 40f);
  vert[5].Color = Color.Blue.ToArgb();
  vert[5].Normal = new Vector3(-1, 0, 1);
}
```

رئوس فوق دو مثلث سه بعدی که از سمت جلو به هم چسبیده و تشکیل یک ۸ داده اند را ایجاد می کند. تنها کار باقی مانده تغییر متد DrawUserPrimitives به شکل زیر است تا دو مثلث بر روی صحنه رسم شود:

```
device.DrawUserPrimitives(PrimitiveType.TriangleList, 2, vert);
```

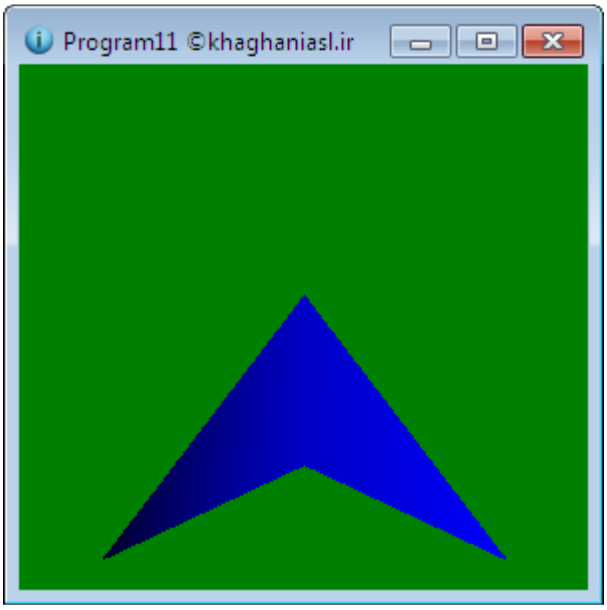
با تغییر رنگ پس زمینه به سبز، پس از اجرای این برنامه، مشاهده خواهید کرد که دو مثلث کاملا مجزا از یکدیگر دیده می شوند:



حال مقدار normal نقاط را به گونه ای تغییر می دهیم که نور همانند شکل b منعکس شود بنابراین متد DrawTriangle را به شکل زیر تغییر می دهیم:

```
public void DrawTriangle()
{
  vert = new CustomVertex.PositionNormalColored[6];
  vert[0].Position = new Vector3(0f, 0f, 30f);
  vert[0].Color = Color.Blue.ToArgb();
  vert[0].Normal = new Vector3(0, 0, 1);
  vert[1].Position = new Vector3(30f, 0f, 0f);
  vert[1].Color = Color.Blue.ToArgb();
  vert[1].Normal = new Vector3(1, 0, 1);
  vert[2].Position = new Vector3(0f, 30f, 30f);
  vert[2].Color = Color.Blue.ToArgb();
  vert[2].Normal = new Vector3(0, 0, 1);
  vert[3].Position = new Vector3(-30f, 0f, 0f);
  vert[3].Color = Color.Blue.ToArgb();
  vert[3].Normal = new Vector3(-1, 0, 1);
  vert[4].Position = new Vector3(0f, 0f, 30f);
  vert[4].Color = Color.Blue.ToArgb();
  vert[4].Normal = new Vector3(0, 0, 1);
  vert[5].Position = new Vector3(0f, 30f, 30f);
  vert[5].Color = Color.Blue.ToArgb();
  vert[5].Normal = new Vector3(0, 0, 1);
}
```

اگر برنامه را اجرا کنید، خواهید دید که نور به سطح مثلثها کاملا طبیعی و نرم تابیده و قسمتهائی که در معرض نور عمود قرار ندارند، به آرامی تاریکتر شده اند.



برای تعیین مقدار normal هر مثلث می توان مختصات دو گوش آن را با یکدیگر جمع و بر ۲ تقسیم کرد. اگر از تکنیک نورپردازی بر روی انبوهی از مثلثهای مختلف (مانند زمین طراحی شده) استفاده کنیم، فضاهای بسیار زیبائی خلق خواهد شد.

توجه نمائید که مثال مطرح شده در این بخش را می توان با استفاده از Index Buffer نیز پیاده سازی کرده و در تعداد رئوس استفاده شده صرفه جوئی نمود چون مختصات نقاط جلوئی دو مثلث و همچنین مقدار normal آنها کاملا با هم برابرند اما برای سادگی از این کار صرف نظر کردیم. البته اگر مایل به داشتن لبه ای باشیم که برشی در

شکل ایجاد کند (حالت اول) مجبوریم از نقاط مجزا استفاده کنیم چون مقدار normal نقاط با هم متفاوت خواهند شد.

۹-۲ ایجاد Mesh

با تعریف فیلد normal برای هر راس مثلث، شیب سطح مثلثهای مختلف رسم شده با یکدیگر متفاوت می شود اما از آنجایی که در یک زمین هر راس، بین ۴ مثلث مشترک است. محاسبه مقدار normal تمام رئوس، یعنی محاسبه میانگین وزنی سایر رئوس آن مثلث، سربار قابل توجهی روی پردازنده خواهد داشت. برای حل این مشکل، نقاط سه بعدی در قالب Mesh باید ارائه شوند.

یک Mesh در حقیقت یک VertexBuffer ادغام شده با یک IndexBuffer با مقداری اطلاعات در مورد نحوه حرکت داخلی شی (در صورت وجود) و نمای ظاهری شی است. مش‌ها می توانند انواع داده گرافیکی را نگهداری کنند ولی معمولا برای کپسوله کردن مدلهای پیچیده به کار می روند. مش‌ها از اجسام اولیه‌ای مثل خط و مثلث تشکیل شده اند و می توانند دارای بافت باشند و تحت نورپردازی قرار گیرند. به عنوان مثال اگر مش، نشان دهنده بدن انسان باشد ممکن است بازو نسبت به بدن حرکت داشته باشد و یا اگر یک خانه را به صورت یک مش در نظر بگیریم یقینا ظاهر سقف از ظاهر دیوارها متفاوت خواهد بود زیرا این دو از مواد مختلفی ساخته شده اند.

بزرگترین ویژگی مش‌ها فراهم کردن امکان استفاده از محصولات سایر تولیدکنندگان اشیا سه بعدی است. برای استفاده از یک شی سه بعدی، کافی است آن را در قالب یک Mesh در برنامه بارگذاری نمود. مش‌ها همچنین دارای توابعی برای بالا بردن سرعت رندر کردن اشیا هستند.

برای استفاده از Mesh، باید فضای نام Microsoft.DirectX.Direct3DX را به رفرنسهای پروژه اضافه کنید. از آنجایی که در Mesh نوع مختصات تمام نقاط short است، مختصات تمام نقاط تعیین شده برای زمین را باید از int به short تغییر دهیم. ابتدا پروژه مربوط به اعمال رنگ بر روی زمین را بارگذاری کنید. حال هر کجایی که از int برای تعریف اندیسها استفاده کرده ایم آن را به short تغییر دهید مثلا هنگام تعریف متغیر indics که آن را باید به صورت زیر تعریف کنیم:

```
private short[] indices;
```

چند تغییر نیز باید در متد IndicesDeclaration انجام دهیم:

```
private void IndicesDeclaration()
{
  ib = new IndexBuffer(typeof(short), (Width - 1)*(Height - 1)*6,
                       device, Usage.WriteOnly, Pool.Default);
  indices = new short[(Width - 1) * (Height - 1) * 6];
  for (int x = 0; x < Width - 1; x++)
  {
    for (int y = 0; y < Height - 1; y++)
    {
     indices[(x+ y*(Width-1))* 6] = (short)((x+1)+ (y+1)*Width);
     indices[(x+ y*(Width-1))* 6+1] = (short)((x+1)+ y*Width);
     indices[(x+ y*(Width-1))* 6+2] = (short)(x+ y*Width);
     indices[(x+ y*(Width-1))* 6+3] = (short)((x+1)+(y+1)*Width);
     indices[(x+ y*(Width-1))* 6+4] = (short)(x+ y*Width);
     indices[(x+ y*(Width-1))* 6+5] = (short)(x+ (y+1)*Width);
    }
  }
  ib.SetData(indices, 0, LockFlags.None);
}
```

متغیری از نوع Mesh در کلاس تعریف می کنیم:

```
private Mesh Mesh1;
```

سپس متدی به نام CreateMesh به شکل زیر تعریف کرده و آن را بلافاصله پس از فراخوانی متدهای VertexDecleration و IndicsDecleration فراخوانی می کنیم:

```
private void CreateMesh()
{
  Mesh1 = new Mesh((Width - 1)*(Height - 1)*2, Width * Height,
   MeshFlags.Managed,CustomVertex.PositionColored.Format,device);
}
```

اولین پارامتر، تعداد مثلثهای موجود در مش و پارامتر دوم، تعداد نقاط موجود در مش را مشخص می کند. پارامتر سوم، وظیفه مدیریت حافظه مربوط به مش را به Managed Environment محول می کند. پارامتر چهارم، نوع نقاط را مشخص کرده و پارامتر آخر هم device می باشد. پس از ایجاد Mesh می توان بافرهای تعریف شده برای زمین را در بافرهای مش کپی کرد. پس دستورات زیر را به متد CreateMesh اضافه می کنیم:

```
Mesh1.SetVertexBufferData(vert, LockFlags.None);
Mesh1.SetIndexBufferData(indices, LockFlags.None);
```

پارامترLockFlags امکان یا عدم امکان اعمال تغییرات بعدی روی بافرها را مشخص می‌کند. LockFlags.None هیچ قفلی بر روی بافرها قرار نمی دهد اما باعث کاهش سرعت می شود.

معمولا DirectX کارها را خیلی بهتر انجام می دهد. متد OptimizeInPlace نقاط تعریف شده در مش را جهت افزایش سرعت، سازماندهی مجدد می کند. این متد برای اجرا به آرایه ای از نوع AdjacencyInformation نیاز دارد. با استفاده از این آرایه، DirectX میزان نزدیکی نقاط نسبت به یکدیگر در مش را متوجه می شود. با استفاده از این اطلاعات DirectX می تواند بهینه سازیهایی از قبیل حذف نقاطی که زیاد به هم نزدیک هستند را انجام دهد. دستورات زیر را نیز به متد CreateMesh اضافه می کنیم:

```
int[] adjac = new int[Mesh1.NumberFaces * 3];
Mesh1.GenerateAdjacency(0.5f, adjac);
Mesh1.OptimizeInPlace(MeshFlags.OptimizeVertexCache, adjac);
```

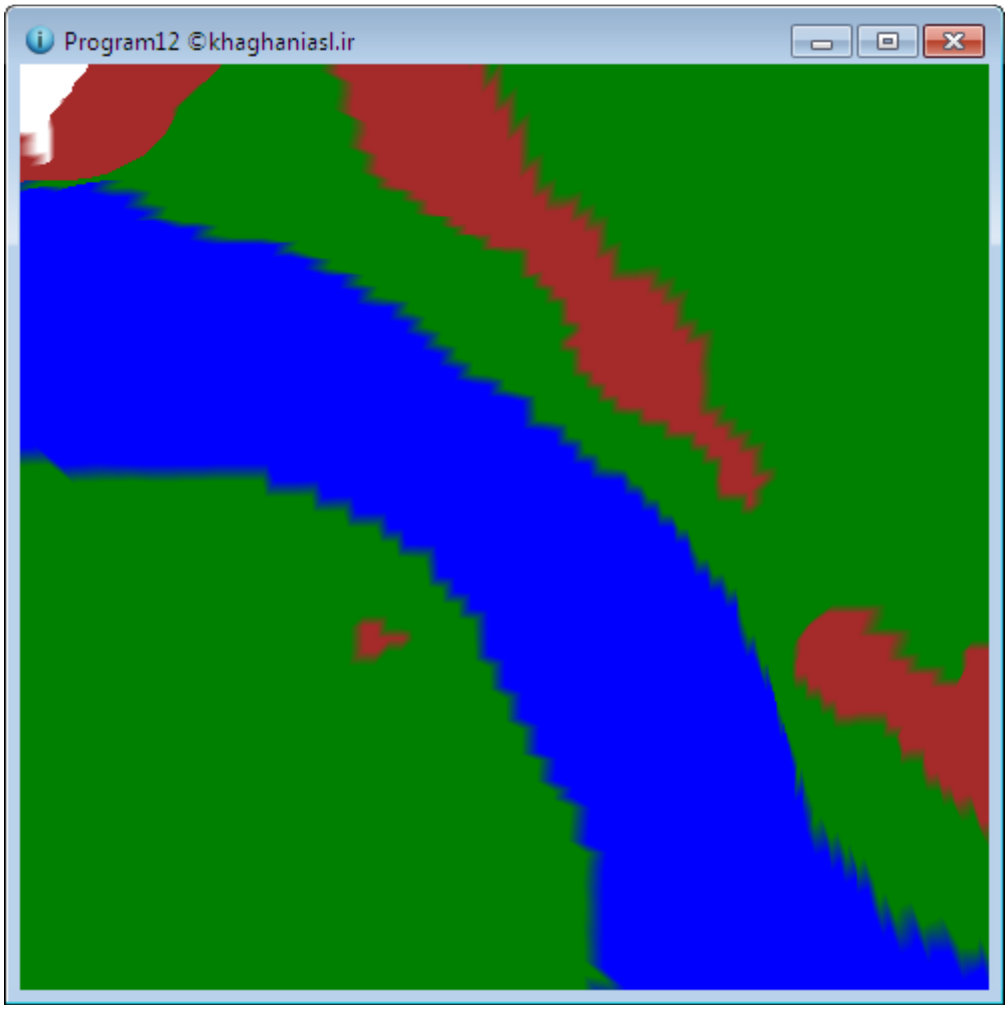
در اولین خط، آرایه ای جهت نگهداری این اطلاعات تعریف شده است. از آنجایی که زمین از مثلثها ساخته شده و هر مثلث با بیش از ۳ نقطه همسایه نیست، مقدار NumberFaces*3 کافی است. در دستور بعدی AdjacencyInformation تولید می شود. در این دستور، به DirectX اعلام شده که هر دو نقطه مجاوری که فاصله ای کمتر از 0.5f داشته باشد، می تواند به عنوان یک نقطه در نظر گرفته شود. البته از آنجایی که در زمین طراحی شده فاصله هر یک از نقاط نسبت به یکدیگر ۱ است، اجرای بهینه سازی تغییری در ساختار زمین نخواهد داشت. در خط بعدی، با تعیین نوع بهینه سازی و ارسال آرایه adjac، متد OptimizeInPlace فراخوانی شده است. انواع مختلف بهینه سازیها را می توان بر روی Mesh انجام داد. نوع بهینه سازی انتخاب شده در مثال فوق باعث کاهش تعداد Cache Missها و در نتیجه افزایش سرعت اجرای برنامه خواهد شد.

حال نوبت رسم Mesh است. کافی است دستورات مابین BeginSence و EndSence را به شکل زیر تغییر دهیم:

```
device.Transform.World = Matrix.Translation(-Height/2,-Width/2,0)
                         * Matrix.RotationZ(angle);
int NumSubSets = Mesh1.GetAttributeTable().Length;
for (int i = 0; i < NumSubSets; ++i)
{
  Mesh1.DrawSubset(i);
}
```

با اجرای برنامه، تغییر محسوسی در زمین مشاهده نخواهید کرد اما اکنون زمین با ترکیب نقاط و اندیسهای تشکیل دهنده آن به صورت یک Mesh ایجاد شده است که متدهای بسیار مفیدی همانند OptimizeInPlace و یا

متدی که شیب سطح هر یک از مثلثها را بصورت اتوماتیک محاسبه می کند (تا نورها به صورت طبیعی بر آنها بتابد) در آن وجود دارد.



علاوه بر این چون زمین هم اکنون با استفاده از یک Mesh رسم می شود، سرعت برنامه افزایش یافته و می توان تمام کدهای مربوط به VertexBuffer و IndicsBuffer را از برنامه حذف کرد.

بسیاری از مش‌ها به تکه های مختلفی تقسیم می شوند و هنوز هم به انتقال و دوران نیاز داریم. به عنوان مثال یک خانه از دیوارها، سقف، کف، در و پنجره ها تشکیل می شود. Mesh ایجاد شده در مثال ما فقط از یک تکه تشکیل شده اما برای حفظ جامعیت، کد لازم برای رسم تمام قسمتهای یک Mesh آورده شده بود.

۲-۹-۱ تابش نور بر زمین

DirectX می تواند بردار normal هر یک از نقاط موجود در Mesh را به صورت اتوماتیک محاسبه کند. بنابراین ابتدا Mesh خود را با مشی که از فرمت PositionNormalColored پشتیبانی می‌کند تعویض می کنیم. این عمل به سادگی با تهیه یک کپی از Mesh (به کمک متد Clone) و تغییر فرمت نقاط آن هنگام ساخت کپی انجام می-شود. کافی است دستورات زیر را به متد CreateMesh اضافه نمائیم:

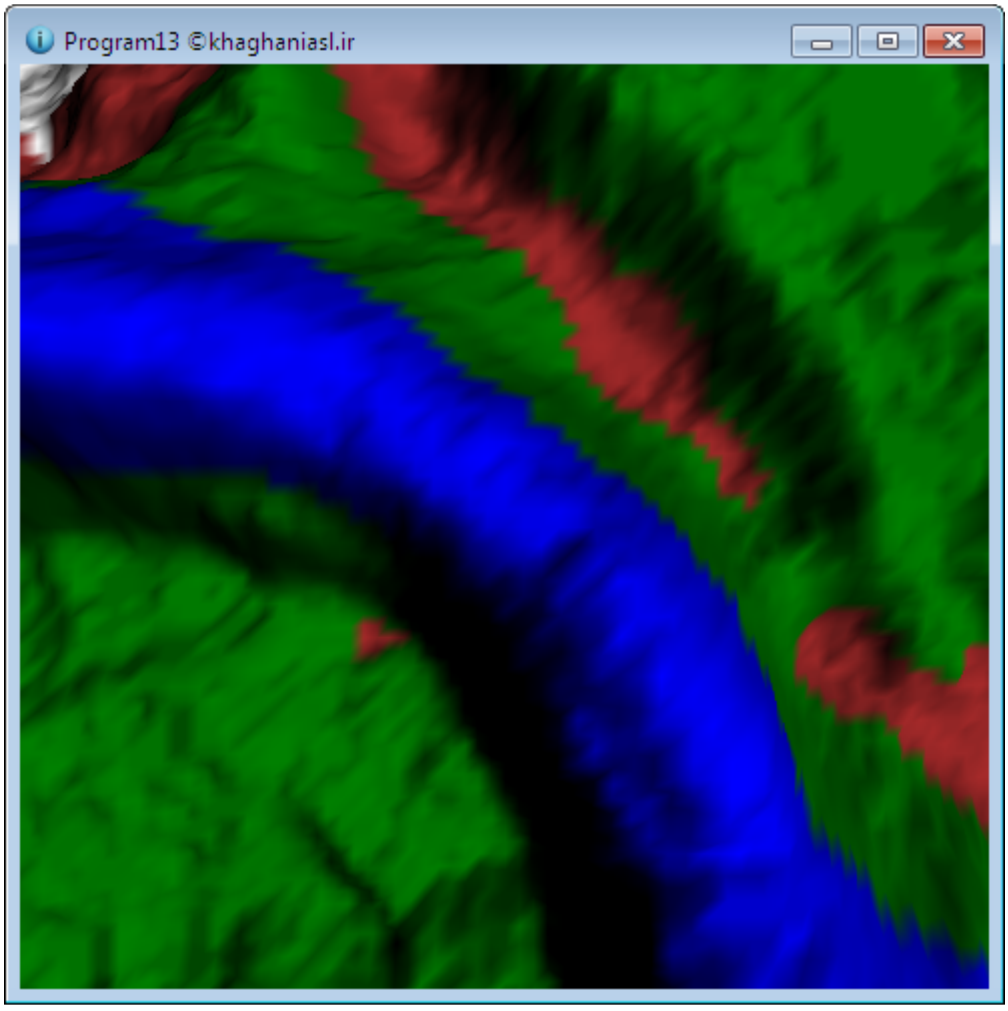
```
Mesh1 = Mesh1.Clone(Mesh1.Options.Value,
                    CustomVertex.PositionNormalColored.Format, device);
Mesh1.ComputeNormals();
```

اولین خط یک کپی از Mesh که از فرمت PositionNormalColored پشتیبانی می کند ایجاد می نماید. اولین پارامتر تمام خصوصیات مش اصلی را در نسخه کپی آن نیز قرار می دهد. دومین پارامتر فرمت جدید نقاط و آخرین پارامتر device می باشد. دومین خط عملیات سنگین لازم برای محاسبه بردار normal هر یک از مثلثهای مش را انجام می دهد.

حال از این Mesh می توان در یک محیط نورپردازی شده استفاده کرده و انعکاس نور از آن را به صورت طبیعی مشاهده نمود. البته هنوز منبع نوری در محیط قرار داده نشده است بنابراین پس از دستورات تعریف دوربین و محل قرارگیری آن، دستورات زیر را اضافه می کنیم:

```
private void Camera()
{
  device.Transform.Projection = Matrix.PerspectiveFovLH(
         (float)Math.PI/4, this.Width/this.Height, 1f, 150f);
  device.Transform.View = Matrix.LookAtLH(new Vector3(0, 0, -50),
                      new Vector3(0, 0, 0), new Vector3(0, 1, 0));
  device.RenderState.Lighting = true;
  device.Lights[0].Type = LightType.Directional;
  device.Lights[0].Diffuse = Color.White;
  device.Lights[0].Direction = new Vector3(-25f, 0, 50f);
  device.Lights[0].Enabled = true;
}
```

در دستورات فوق یک منبع نور مستقیم در مختصات (۲۵,۰,۵۰-) با رنگ سفید قرار داده شده و فعال می شود. پس از اجرای برنامه، محیطی بسیار طبیعی تر از قبل ظاهر خواهد شد.



فصل سوم: ساخت محیطهای سه بعدی

در فصول اول و دوم، برخی مفاهیم پایه ای در گرافیک و پیاده سازی آنها توسط DirectX بررسی شد. در این فصل اصول ایجاد یک محیط سه بعدی کامل ارائه خواهد شد، به طوری که پس از پایان فصل قادر به ساخت محیطها و بازیهای سه بعدی دلخواهتان خواهید بود.

تمام مطالب این فصل در قالب ساخت یک بازی سه بعدی که پرواز هواپیما را شبیه سازی می کند، آموزش داده خواهد شد. در این بازی خواهیم توانست هواپیمائی را بر فراز یک شهر به پرواز در آورده و افکتهای صوتی و متنی به محیط اضافه کرد. البته بدیهی است هدف از ساخت این بازی صرفا آموزش مفاهیم DirectX بوده و نباید بازی ساخته شده را با بازیهای سه بعدی و زیبای موجود که تمام قوانین فیزیک و جاذبه و... در آنها رعایت می شود، مقایسه کرد. در ساخت بازیهای امروزی علاوه بر استفاده از پیشرفته ترین تکنیکهای DirectX از محاسبات ریاضیاتی و فیزیکی بسیار سنگینی نیز استفاده می شود که آموزش این مطالب، خود نیاز به کتاب مستقلی دارد.

لیست موضوعاتی که در این فصل به آنها پرداخته خواهد شد به شرح زیر هستند:

- افزودن Texture به مثلثها
- ایجاد داینامیک محیط شهر
- ایجاد SkyBox و رهائی از پس زمینه تاریک
- تکنیکهای پایه ای برای مدل سازی پرواز
- استفاده از صفحه کلید و ماوس به کمک DirectInput
- استفاده از مفهوم برخورد
- پخش افکتهای صوتی و صدا در پس زمینه
- نمایش متن در صحنه

۳-۱ ایجاد پروژه

با استفاده از دانش کسب شده در فصول قبلی، یک پروژه جدید ایجاد کرده و عملیاتهای زیر را انجام دهید:

۱- متصل شدن به کارت گرافیکی و پیکر بندی device
۲- ایجاد ZBuffer و پاک کردن آن
۳- تنظیم دوربین، محل قرار گیری آن و محل نگرش آن

۳-۱-۱ استفاده از Texture

در بخشهای قبلی برای ایجاد صحنه های رنگی از تکنیک تخصیص رنگ به نقاط تعریف شده استفاده شد. بدیهی است که برای خلق محیطهای رنگی در بازیهای امروزی از این تکنیک استفاده نمی شود. DirectX از یک تکنیک بسیار قدرتمند برای افزودن تصاویر به محیط استفاده می کند. به سادگی می توان یک تصویر را بر روی یک مثلث کشید. به چنین تصاویری در اصطلاح Texture گفته می شود.

به عنوان مثال، می خواهیم مثلثی را رسم کرده و تصویری را روی آن بکشیم. ابتدا آرایه ای از نقاط برای ایجاد یک مثلث در کلاس تعریف کرده و فرمت نقاط را از نوع PositionTextured تعیین می کنیم:

```
private CustomVertex.PositionTextured[] vertices;
```

سپس در متد VertexDeclaration، مختصات سه نقطه مثلث را تعریف می کنیم:

```
private void VertexDeclaration()
{
  vertices = new CustomVertex.PositionTextured[3];
  vertices[0].Position = new Vector3(10f, 10f, 0f);
  vertices[0].Tu = 0;
  vertices[0].Tv = 0;
  vertices[1].Position = new Vector3(-10f, -10f, 0f);
  vertices[1].Tu = 1;
  vertices[1].Tv = 1;
  vertices[2].Position = new Vector3(10f, -10f, 0f);
  vertices[2].Tu = 0;
  vertices[2].Tv = 1;
}
```

در متد بالا، علاوه بر تعیین مختصات هر نقطه، دو فیلد Tu و Tv نیز برای هر نقطه مقداردهی شده است. این دو فیلد برای منطبق کردن تصویر روی مثلث استفاده می شود. به عنوان مثال، اگر مقدار این دو فیلد برای نقطه ای (۰,۰) انتخاب شود، گوشه سمت چپ بالای تصویر در آن نقطه قرار خواهد گرفت. سایر مقادیری که می توان برای این دو فیلد در نظر گرفت، مقادیر (۱,۰) برای گوشه سمت راست بالا، (۰,۱) گوشه سمت چپ پائین و (۱,۱) برای گوشه سمت راست پائین می باشد. همانند قبل متد VertexDeclaration بایستی در متد سازنده (و یا هر جای مناسب دیگری) فراخوانی گردد.

جهت رسم مثلث، کافی است دستورات زیر را بین BeginScene و EndScene در متد OnPaint قرار دهیم:

```
device.VertexFormat = CustomVertex.PositionTextured.Format;
device.DrawUserPrimitives(PrimitiveType.TriangleList,1,vertices);
```

با اجرای برنامه تا این لحظه، یک مثلث سفید و خالی مشاهده خواهد شد چون هنوز تصویری بر روی آن کشیده نشده است. برای کشیدن یک تصویر بر روی مثلث ابتدا دو متغیر در کلاس، یکی برای نگهداری تصویر و دیگری برای نگهداری نوع Matrial شی، تعریف می کنیم:

```
private Texture texture;
private Material material;
```

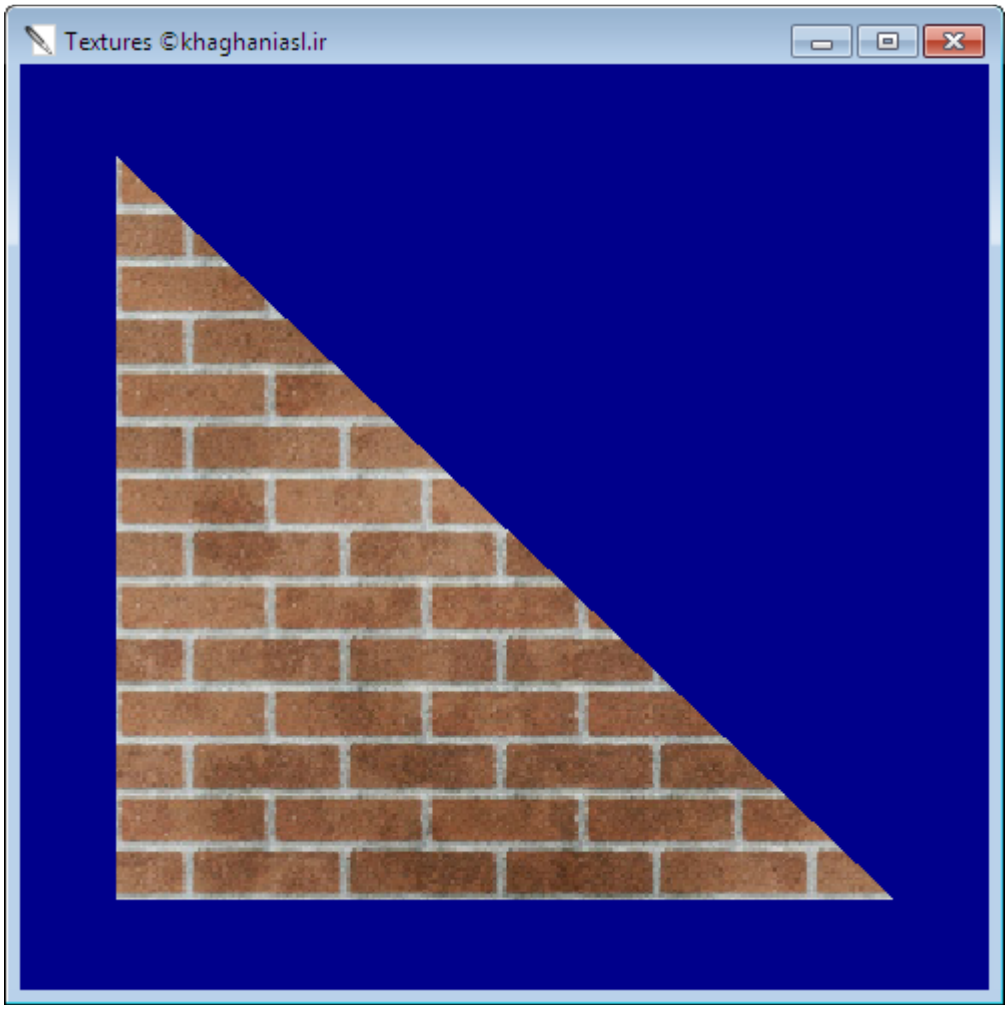
حال متد LoadTexturesAndMaterials را جهت تعیین تصویر texture و نوع matrial جاری سیستم به شکل زیر تعریف می کنیم:

```
private void LoadTexturesAndMaterials()
{
  material = new Material();
  material.Diffuse = Color.White;
  material.Specular = Color.LightGray;
  material.SpecularSharpness = 15.0F;
  device.Material = material;
  texture = TextureLoader.FromFile(device, "Picture.bmp");
  device.SetTexture(0, texture);
}
```

در متد فوق، ابتدا یک شی Matrial ایجاد کرده و برخی خصوصیات آن را مقداردهی می کنیم. در واقع Matrial یک مثلث، نحوه انعکاس نور از سطح مثلث را مشخص می کند. هر Matrial نیز دو نوع نور را منعکس می کند، Diffuse و Specular. البته این دو خصوصیت فقط زمانی که در محیط منبع نوری داشته باشیم موثر خواهند بود در غیراین صورت مقداردهی آنها هیچ تاثیری در خروجی نخواهد داشت. خط چهارم شدت انعکاس نور را مشخص می کند. در خط پنجم Matrial ساخته شده به عنوان Matrial جاری سیستم تنظیم شده است. در خط بعدی، تصویری از یک فایل bmp به شی texture بارگذاری شده و در خط آخر، این شی به عنوان texture جاری سیستم

انتخاب شده است. برای استفاده از متد TextureLoader بایستی فضای نام Microsoft.DirectX.Direct3DX را به رفرنسهای پروژه اضافه کنید.

حال کافی است متد LoadTexturesAndMaterials در متد سازنده فراخوانی گردد. با اجرای برنامه، نصف تصویر بارگذاری شده بر روی مثلث رسم خواهد شد:



برای رسم کل تصویر، دو مثلث تعریف کرده و نیمه دوم تصویر را بر روی مثلث دیگر می کشیم. بنابراین متد VertexDecleration را به شکل زیر تغییر می دهیم:

```
private void VertexDeclaration()
{
  vertices = new CustomVertex.PositionTextured[6];
  vertices[0].Position = new Vector3(10f, 10f, 0f);
  vertices[0].Tu = 0;
  vertices[0].Tv = 0;
  vertices[1].Position = new Vector3(-10f, -10f, 0f);
  vertices[1].Tu = 1;
  vertices[1].Tv = 1;
  vertices[2].Position = new Vector3(10f, -10f, 0f);
  vertices[2].Tu = 0;
  vertices[2].Tv = 1;
  vertices[3].Position = new Vector3(-10.1f, -9.9f, 0f);
  vertices[3].Tu = 1;
  vertices[3].Tv = 1;
  vertices[4].Position = new Vector3(9.9f, 10.1f, 0f);
```

```
  vertices[4].Tu = 0;
  vertices[4].Tv = 0;
  vertices[5].Position = new Vector3(-10.1f, 10.1f, 0f);
  vertices[5].Tu = 1;
  vertices[5].Tv = 0;
}
```

به سادگی یک مثلث دیگر را تعریف کرده و مختصات Tu و Tv آن را به گونه ای تعیین کرده ایم که نیمه دوم تصویر بر روی آن قرار گیرد. البته افزایش تعداد مثلثها را در تابع DrawUserPrimitives به اطلاع DirectX نیز بایستی برسانید:

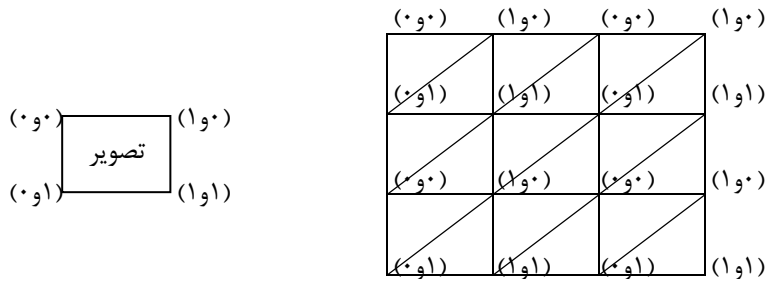
```
device.DrawUserPrimitives(PrimitiveType.TriangleList,2,vertices);
```

حال با اجرای برنامه، کل تصویر باید بر روی دو مثلث کشیده شود:

البته مختصات نقاط دو مثلث به گونه ای داده شده است که فاصله ای بین دو مثلث وجود داشته باشد تا بتوان رسم تصویر بر روی دو مثلث را تشخیص داد. می توان با تغییر مختصات نقاط، این فاصله را حذف و شکلی یکپارچه به وجود آورد. همچنین با تغییر مقدار Tu یا Tv نقاط می توان جهت کشیده شدن شکل بر روی مثلثها را تغییر داد. برای فیلدهای Tu یا Tv می توان هر مقداری بین صفر تا یک را انتخاب کرد.

۳-۲ ترسیم سطح زمین

در این بخش نحوه نشان دادن تصاویر ساده بر روی مجموعه ای از مثلثها بررسی خواهد شد. مهمترین کار برای نشان دادن یک یا چند تصویر بر روی مجموعه ای از مثلثها، تعیین مختصات و ویژگیهای نقاط است. می خواهیم یک texture را بر روی زمین بکشیم:



معمولا texture ها به صورت قرینه طراحی می شوند یعنی چه از سمت راست و چه از سمت چپ رسم شوند، یکسان دیده می شوند. بنابراین برای اینکه Tu و Tvهای اندیسها با هم تداخل پیدا نکند، از انعکاس texture ها استفاده می شود.

در سطوح پهناور که از اندیسها برای ایجاد مثلثها استفاده می شود، از آنجایی که مثلثها به صورت اشتراکی از نقاط مختلف استفاده می کنند، تعیین یک مقدار برای Tu و Tv برای تمام مثلثها صحیح نخواهد بود. در این صورت باید texture مربع بعدی به صورت قرینه نسبت به مربع قبلی رسم گردد یعنی Tu و Tv مثلثها هم از لحاظ ستونی و هم از لحاظ سطری به گونه ای انتخاب می شوند که texture رسم شده بر روی آنها مکمل یکدیگر شود.

فرض کنید می خواهیم یک سطح مربعی ۳*۳ که خانه وسطی آن فاقد تصویر است، ایجاد کنیم. این سطح دارای ۸ مربع حاوی تصویر، و بنابراین ۱۶ مثلث و در نتیجه ۴۸ نقطه خواهد بود. برای تعریف این سطح، فضای نام System.Collections را به پروژه و متغیرهای زیر را به کلاس اضافه می کنیم:

```
private int[,] FloorPlan;
private int Width;
private int Height;
private int Buildings = 5;
private CustomVertex.PositionNormalTextured[] VerticesArray;
ArrayList VerticesList = new ArrayList();
```

دو متغیر Height و Width طول و عرض سطح را نگهداری خواهند کرد. متغیر Buildings نیز انواع ساختمانهائی که بر روی زمین وجود خواهد داشت را مشخص می کند. آرایه VerticesArray نیز نقاط لازم برای ایجاد مثلثها را نگهداری خواهد کرد. همانطور که ملاحظه می‌کنید برای نگهداری نقاط از ArrayList استفاده کرده‌ایم. این نوع داده، مشابه آرایه است با این تفاوت که لازم نیست حداکثر اندازه آن را از قبل مشخص کنیم، از این رو برای چنین حالتهائی که معلوم نیست چند نقطه ایجاد خواهد شد، بسیار مناسب است. تعداد داده های افزوده شده به ArrayList را می توان از طریق فیلد count آن به دست آورد. علاوه بر تعیین مختصات نقاط، مقدار normal هر یک از آنها نیز محاسبه خواهد شد تا در بخشهای بعدی که از نور استفاده خواهیم کرد، مشکلی رخ ندهد.

با استفاده از متد LoadFloorPlan نقشه اولیه سطح مورد نظر را در آرایه FloorPlan ذخیره می کنیم:

```
private void LoadFloorPlan()
{
  Width = 3;
  Height = 3;
  FloorPlan = new int[,]{{0,0,0},{0,1,0},{0,0,0}};
}
```

در ساختمان داده فوق، وجود 0 نشان دهنده رسم مربع و کشیدن تصویر بر روی آن در ناحیه متناظر و وجود 1 نیز به معنی خالی بودن ناحیه متناظر است. در بخش بعد وجود 1 به معنی وجود ساختمان در آن ناحیه و وجود 0 به معنی سطح در ناحیه متناظر است. این ساختمان داده قابلیت انعطاف برنامه را افزایش می دهد. مثلا برای افزودن یک ساختمان جدید به محیط کافی است یکی از صفرها را به ۱ تغییر دهیم. این متد را بایستی در داخل تابع سازنده قبل از هر متد دیگری فراخوانی کنیم.

می توان با ادغام تمام تصاویری که بایستی به عنوان در صحنه استفاده شوند در یک فایل، صحنه ای با textureهای مختلف ایجاد کرد. از تصویری برای ساخت محیط خود استفاده می کنیم که از چندین تصویر کوچک تشکیل شده و ما هر یک از این تصاویر کوچک را بر روی ناحیه خاصی خواهیم کشید. مثلا چپ ترین تصویر برای کف زمین، تصویر بعدی برای دیوار ساختمانها و تصاویر بعدی برای سقف ساختمانهای مختلف استفاده خواهد شد.

در ادامه متد LoadTexturesAndMaterials را کمی تغییر داده و تصویر فوق در فایل را بارگذاری می کنیم:

```
private void LoadTexturesAndMaterials()
{
  material = new Material();
  material.Diffuse = Color.White;
  material.Ambient = Color.White;
  device.Material = material;
  texture = TextureLoader.FromFile(device, "MultiTexture.bmp");
  device.SetTexture(0, texture);
}
```

همانطور که ملاحظه می کنید این فایل حاوی یک تصویر برای سطح زمین و ۲ تصویر برای هر یک از ۵ نوع ساختمان است. بنابراین در متد VertexDeclaration ابتدا متغیری به شکل زیر تعریف می کنیم:

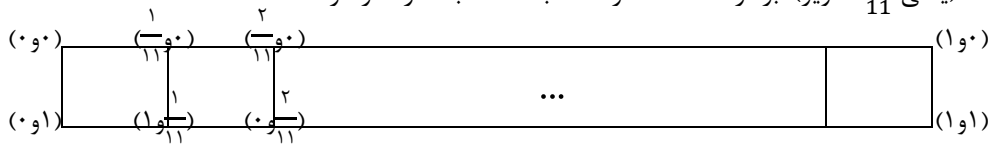
```
float ImagesInTexture = 1 + Buildings * 2;
```

حال باید در ادامه این متد، آرایه FloorPlan را پیمایش کرده و با مشاهده هر صفر، دو مثلث ساخته و تصویر اول موجود در Texture را بر روی آنها بکشیم:

```
for (int x = 0; x < Width; x++)
{
  for (int y = 0; y < Height; y++)
  {
    if (FloorPlan[x, y] == 0)
    {
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x,y,0), new Vector3(0,0,1), 1f/ImagesInTexture, 1));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x+1,y,0), new Vector3(0,0,1), 0, 1));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x+1,y+1,0), new Vector3(0,0,1), 0, 0));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x+1,y+1,0), new Vector3(0,0,1), 0, 0));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x,y+1,0), new Vector3(0,0,1), 1f/ImagesInTexture,0));
```

```
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x,y,0), new Vector3(0,0,1), 1f/ImagesInTexture, 1));
    }
  }
}
```

در کد فوق در صورت مشاهده هر صفر، دو مثلث ساخته شده و ناحیه مناسب از texture (تصویر اول در texture) بر روی آن کشیده می شود. این ناحیه در حقیقت مستطیلی از مختصات (0,0) تا (1f/ImagesInTexture,1)، می‌باشد. (یعنی $\frac{1}{11}$ تصویر). بردار normal هر نقطه به سمت بالا در نظر گرفته شده است.



پس از افزودن تمام نقاط به ArrayList با استفاده از متد ToArray، می توان داده های درج شده به آن را در قالب یک آرایه عادی به دست آورد. برای این منظور فرمت هر خانه آرایه را باید به متد ToArray اعلام کنیم:

```
VerticesArray =
(CustomVertex.PositionNormalTextured[])VerticesList.ToArray(
                    typeof(CustomVertex.PositionNormalTextured));
```

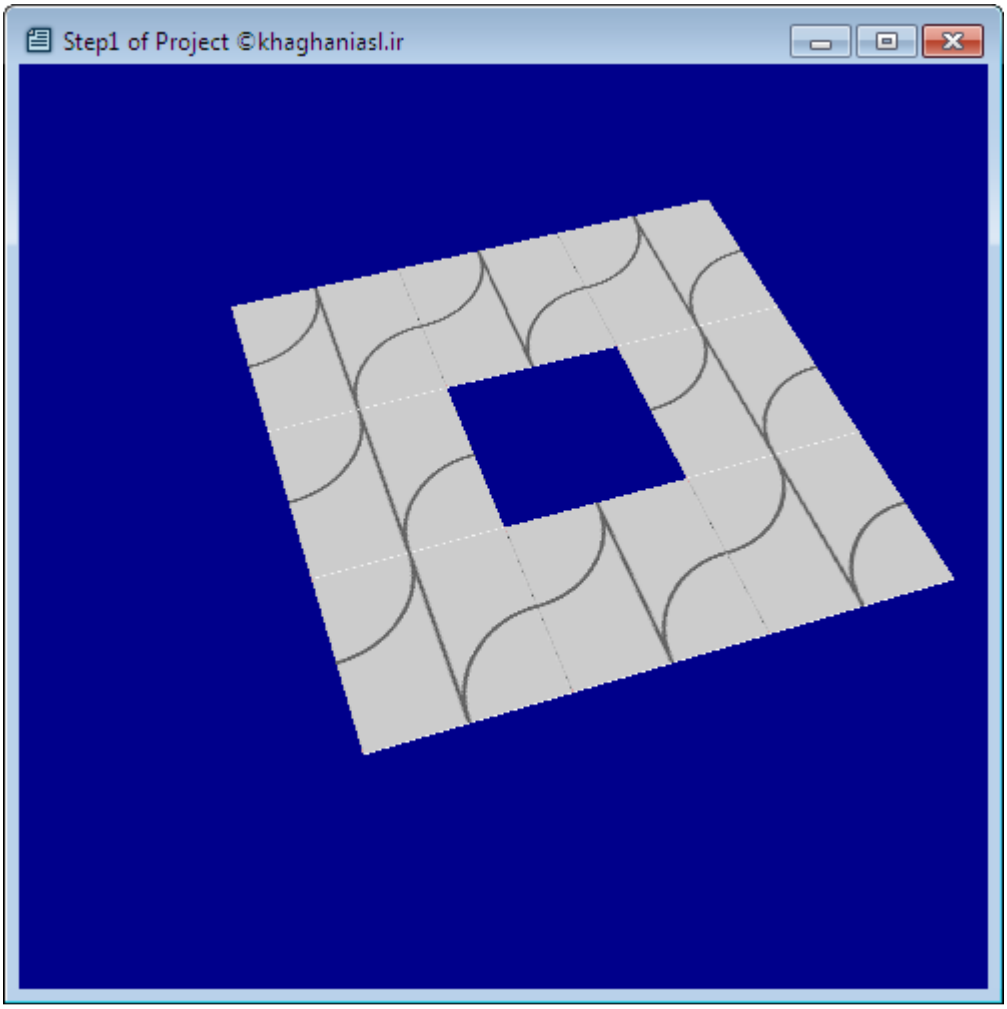
تنها کار باقی مانده اعلام تغییر نوع فرمت نقاط به DirectX و تغییر متد DrawUserPrimitives جهت نمایش مثلثهای جدید از روی آرایه VerticesArray است. بنابراین دستورات زیر را بین BeginScene و EndScene در متد OnPaint قرار می دهیم:

```
device.VertexFormat = CustomVertex.PositionNormalTextured.Format;
device.DrawUserPrimitives(PrimitiveType.TriangleList,
                          VerticesList.Count/3, VerticesArray);
```

اگر مختصات دوربین را کمی تغییر دهیم:

```
device.Transform.View = Matrix.LookAtLH(new Vector3(3, -2, 5),
                      new Vector3(2, 1, 0), new Vector3(0, 0, 1));
```

با اجرای برنامه، مربعی با نمای سه بعدی را خواهید دید که وسط آن خالی است. اگر قبل از رسم مثلثها، دستور device.RenderState.FillMode = FillMode.WireFrame; را در متد OnPaint درج کنید، ساختار مثلثها بدون پوشش تصویریشان نمایان خواهد شد.



۳-۳ ترسیم ساختمانها

قصد داریم ۵ ساختمان روی سطح زمین ترسیم کنیم. پس آرایه ای برای نگهداری ارتفاع هر یک از ساختمانها تعریف و به ابتدای کلاس اضافه می کنیم:

```
private int[] BuildingHeights = new int[] {0, 10, 1, 3, 2, 5};
```

سپس ماتریسی برای سطح زمین ایجاد می کنیم که درایه های آن اعداد بین صفر تا ۵ است. صفرها به معنی عدم وجود ساختمان و کف زمین بوده و هر یک از اعداد ۱ تا ۵ نیز نشان دهنده نوع ساختمان در محل متناظر خواهند بود. این اعداد بصورت تصادفی روی ماتریس توزیع خواهند شد. برای ایجاد این ماتریس کد زیر را به انتهای متد LoadFloorPlan اضافه می کنیم:

```
Random random = new Random();
for (int x = 0; x < Width; x++)
{
  for (int y = 0; y < Height; y++)
    if (FloorPlan[x, y] == 1)
      FloorPlan[x, y] = random.Next(Buildings) + 1;
}
```

ابتدا یک شی تولیدکننده اعداد تصادفی ایجاد شده است که با فراخوانی متد Next آن یک عدد تصادفی تولید شود. سپس کل درایه های ماتریس بررسی شده و به ازای هر درایه ۱، یک عدد تصادفی بین ۱ تا ۵ با آن جایگزین خواهد شد.

متد VertexDeclaration را نیز به شکل زیر تغییر می دهیم:

```
for (int x = 0; x < Width; x++)
{
  for (int y = 0; y < Height; y++)
  {
    int CurrentBuilding = FloorPlan[x, y];
    VerticesList.Add(new CustomVertex.PositionNormalTextured(new
Vector3(x,y, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), (CurrentBuilding*2 + 1)/ImagesInTexture, 1));
    VerticesList.Add(new CustomVertex.PositionNormalTextured(new
Vector3(x+1,y, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), CurrentBuilding*2/ImagesInTexture, 1));
    VerticesList.Add(new CustomVertex.PositionNormalTextured(new
Vector3(x+1,y+1, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), CurrentBuilding*2/ImagesInTexture, 0));
    VerticesList.Add(new CustomVertex.PositionNormalTextured(new
Vector3(x+1,y+1, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), CurrentBuilding*2/ImagesInTexture, 0));
    VerticesList.Add(new CustomVertex.PositionNormalTextured(new
Vector3(x,y+1, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), (CurrentBuilding*2+1)/ImagesInTexture, 0));
    VerticesList.Add(new CustomVertex.PositionNormalTextured(new
Vector3(x,y, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), (CurrentBuilding*2+1)/ImagesInTexture, 1));
  }
}
```

مختصات z تمام نقاط از ماتریسی که در بالا ایجاد کردیم، اخذ شده است. مختصات x برای texture به صورت داینامیک محاسبه می شود. از آنجایی که ۵ نوع ساختمان داریم، مختصات y برای سمت چپ کف زمین برابر $\frac{0}{11}$، برای سقف ساختمان اول برابر $\frac{2}{11}$، برای سقف ساختمان دوم برابر $\frac{4}{11}$ ... و برای سقف ساختمان پنجم برابر $\frac{10}{11}$ می-باشد که فرمول $\frac{\text{ساختمان جاری}*2}{\text{تعداد تصاویر موجود در بافت}}$ را نتیجه خواهد داد.

مختصات y برای سمت راست کف زمین برابر $\frac{1}{11}$ برای سقف ساختمان اول برابر $\frac{3}{11}$ ... و برای سقف ساختمان آخر برابر $\frac{11}{11}$ است که فرمول $\frac{\text{ساختمان جاری}*2 + 1}{\text{تعداد تصاویر موجود در بافت}}$ را نتیجه می دهد. حال با مشاهده هر صفر، تصویر مربوط به کف زمین و با مشاهده هر عدد غیرصفر، تصویر مربوط به سقف ساختمان مربوطه در ارتفاع لازم رسم می شود.

تا اینجا تقریبا تمام مفاهیم لازم برای رسم بافت بر روی اشیا سه بعدی مطرح شد. برای رسم تصویر دیوارها، کافی است دستوراتی به متد VertexDeclaration اضافه کنیم:

```
private void VertexDeclaration()
{
  float ImagesInTexture = 1 + Buildings * 2;
  for (int x = 0; x < Width; x++)
  {
    for (int y = 0; y < Height; y++)
    {
      int CurrentBuilding = FloorPlan[x, y];
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x, y, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), (CurrentBuilding * 2 + 1) / ImagesInTexture, 1));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x + 1, y, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), CurrentBuilding * 2 / ImagesInTexture, 1));
```

```
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x + 1, y + 1, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), CurrentBuilding * 2 / ImagesInTexture, 0));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x + 1, y + 1, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), CurrentBuilding * 2 / ImagesInTexture, 0));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x, y + 1, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), (CurrentBuilding * 2 + 1) / ImagesInTexture, 0));
      VerticesList.Add(new CustomVertex.PositionNormalTextured(
new Vector3(x, y, BuildingHeights[CurrentBuilding]), new
Vector3(0,0,1), (CurrentBuilding * 2 + 1) / ImagesInTexture, 1));
```

```
      if (y > 0)
      {
       if (FloorPlan[x, y - 1] != FloorPlan[x, y])
       {
         if (FloorPlan[x, y - 1] > 0)
         {
           CurrentBuilding = FloorPlan[x, y - 1];
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x,y,0f), new Vector3(0, 1, 0),
           (CurrentBuilding * 2 - 1) / ImagesInTexture, 1));
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x+1,y,BuildingHeights[CurrentBuilding]),new
           Vector3(0,1,0),CurrentBuilding*2/ImagesInTexture,0));
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x+1,y,0f), new Vector3(0, 1, 0),
           CurrentBuilding*2/ImagesInTexture, 1));
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x+1,y,BuildingHeights[CurrentBuilding]), new
           Vector3(0,1,0),CurrentBuilding*2/ImagesInTexture, 0));
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x,y,0f), new Vector3(0,1,0),
           (CurrentBuilding*2-1)/ImagesInTexture, 1));
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x,y,BuildingHeights[CurrentBuilding]), new
           Vector3(0,1,0),(CurrentBuilding*2-1)/ImagesInTexture,
           0));
         }
         if (FloorPlan[x, y] > 0)
         {
           CurrentBuilding = FloorPlan[x, y];
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x, y, 0f), new Vector3(0, -1, 0),
           CurrentBuilding * 2 / ImagesInTexture, 1));
           VerticesList.Add(new
           CustomVertex.PositionNormalTextured(new
           Vector3(x + 1, y, 0f), new Vector3(0, -1, 0),
           (CurrentBuilding * 2 - 1) / ImagesInTexture, 1));
           VerticesList.Add(new
```

```
        CustomVertex.PositionNormalTextured(new
        Vector3(x + 1, y, BuildingHeights[CurrentBuilding]),
        new Vector3(0,-1,0),
        (CurrentBuilding*2-1)/ImagesInTexture, 0));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y, 0f), new Vector3(0, -1, 0),
        CurrentBuilding*2/ImagesInTexture, 1));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x + 1, y, BuildingHeights[CurrentBuilding]),
        new Vector3(0, -1, 0),
        (CurrentBuilding * 2 - 1) / ImagesInTexture, 0));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y, BuildingHeights[CurrentBuilding]), new
        Vector3(0, -1, 0),
        CurrentBuilding * 2 / ImagesInTexture, 0));
      }
     }
    }
    if (x > 0)
    {
     if (FloorPlan[x - 1, y] != FloorPlan[x, y])
     {
      if (FloorPlan[x - 1, y] > 0)
      {
        CurrentBuilding = FloorPlan[x - 1, y];
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x,y,0f), new Vector3(1, 0, 0),
        CurrentBuilding * 2 / ImagesInTexture, 1));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y + 1, 0f), new Vector3(1, 0, 0),
        (CurrentBuilding * 2 - 1) / ImagesInTexture, 1));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y + 1, BuildingHeights[CurrentBuilding]),
        new Vector3(1, 0, 0),
        (CurrentBuilding * 2 - 1) / ImagesInTexture, 0));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y + 1, BuildingHeights[CurrentBuilding]),
        new Vector3(1, 0, 0),
        (CurrentBuilding * 2 - 1) / ImagesInTexture, 0));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y, BuildingHeights[CurrentBuilding]), new
        Vector3(1,0,0),CurrentBuilding*2/ImagesInTexture,0));
        VerticesList.Add(new
        CustomVertex.PositionNormalTextured(new
        Vector3(x, y, 0f), new Vector3(1, 0, 0),
        CurrentBuilding * 2 / ImagesInTexture, 1));
      }
      if (FloorPlan[x, y] > 0)
      {
        CurrentBuilding = FloorPlan[x, y];
```

```
                VerticesList.Add(new
                CustomVertex.PositionNormalTextured(new
                Vector3(x, y, 0f), new Vector3(-1, 0, 0),
                (CurrentBuilding * 2 - 1) / ImagesInTexture, 1));
                VerticesList.Add(new
                CustomVertex.PositionNormalTextured(new
                Vector3(x, y, BuildingHeights[CurrentBuilding]), new
                Vector3(-1,0,0),
                (CurrentBuilding*2-1)/ImagesInTexture, 0));
                VerticesList.Add(new
                CustomVertex.PositionNormalTextured(new
                Vector3(x, y + 1, 0f), new Vector3(-1, 0, 0),
                CurrentBuilding * 2 / ImagesInTexture, 1));
                VerticesList.Add(new
                CustomVertex.PositionNormalTextured(new
                Vector3(x, y, BuildingHeights[CurrentBuilding]), new
                Vector3(-1,0,0),
                (CurrentBuilding*2-1)/ImagesInTexture, 0));
                VerticesList.Add(new
                CustomVertex.PositionNormalTextured(new
                Vector3(x, y + 1, BuildingHeights[CurrentBuilding]),
                new Vector3(-1, 0, 0),
                CurrentBuilding * 2 / ImagesInTexture, 0));
                VerticesList.Add(new
                CustomVertex.PositionNormalTextured(new
                Vector3(x, y + 1, 0f), new Vector3(-1, 0, 0),
                CurrentBuilding * 2 / ImagesInTexture, 1));
            }
          }
        }
      }
    }
    VerticesArray = (CustomVertex.PositionNormalTextured[])
    VerticesList.ToArray(typeof
    (CustomVertex.PositionNormalTextured));
}
```

اگرچه این متد نسبتا طولانی است اما چیز جدیدی در آن وجود ندارد. در ابتدا کف و سقفها رسم شده اند. سپس قسمتهائی از دو ساختمان مختلف که دارای دیوار مشترک هستند، رسم می شوند.

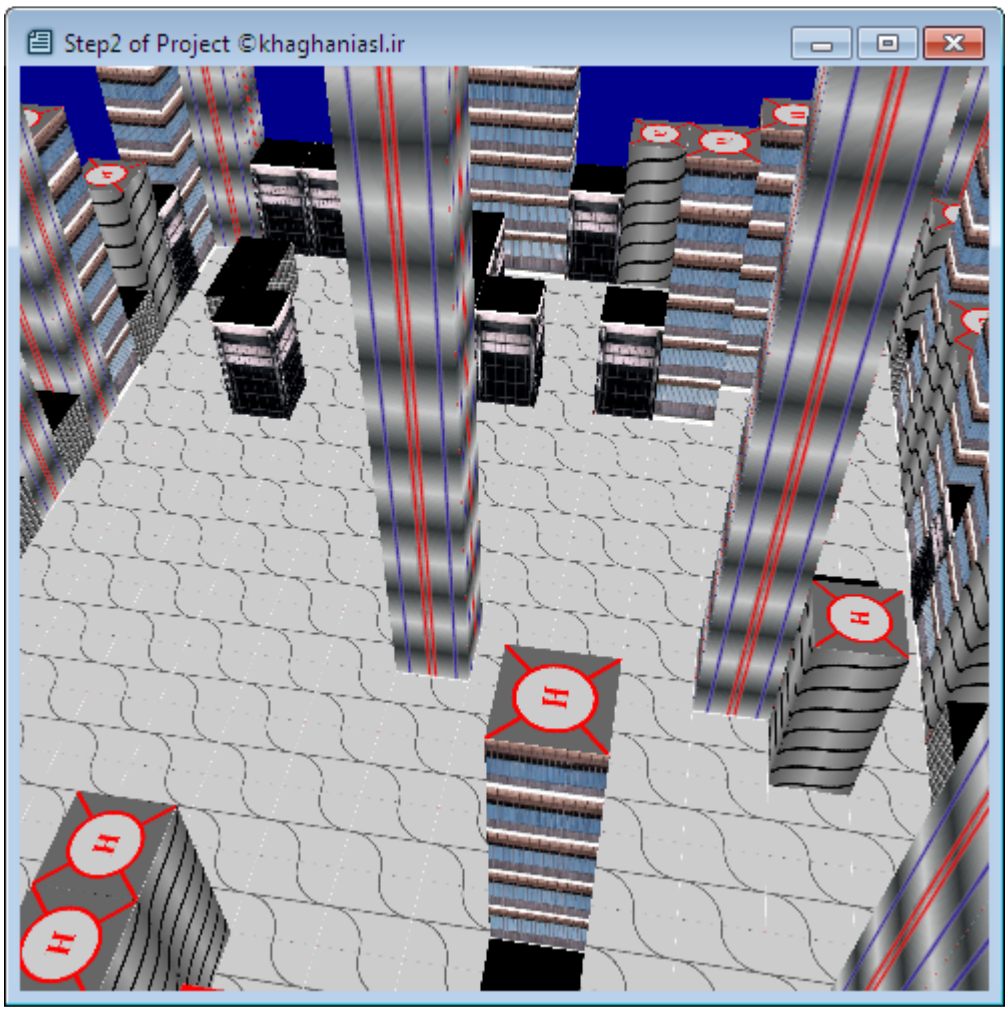
برای ساختن یک شهر بزرگتر، در متد LoadFloorPlan، آرایه FloorPlan را به شکل زیر تغییر می دهیم:

```
Width = 20;
Height = 15;
FloorPlan = new int[,]{
 {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,1,1,0,0,0,1,1,0,0,1,0,1},{1,0,0,1,1,0,0,0,1,0,0,0,1,0,1},
 {1,0,0,0,1,1,0,1,1,0,0,0,0,0,1},{1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,1,1,0,0,0,1,0,0,0,0,0,0,1},{1,0,1,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,0,0,0,0,0,0,0,0,0,1},{1,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
 {1,0,0,0,0,1,0,0,0,0,0,0,0,0,1},{1,0,0,0,0,1,0,0,0,1,0,0,0,0,1},
 {1,0,1,0,0,0,0,0,0,1,0,0,0,0,1},{1,0,1,1,0,0,0,0,1,1,0,0,0,1,1},
 {1,0,0,0,0,0,0,0,1,1,0,0,0,1,1},{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
```

از آنجایی که شهر بزرگتر شده است، موقعیت دوربین را کمی تغییر می دهیم:

```
device.Transform.View = Matrix.LookAtLH(new Vector3(20, 5, 13),
                        new Vector3(8,7,0), new Vector3(0,0,1));
```

حال با اجرای برنامه، تصویری مشابه شکل زیر مشاهده خواهید کرد:



۳-۳-۱ ایجاد Mesh از فایلهای سه بعدی

در DirectX در حالت کلی سه نوع Mesh می توان ایجاد کرد:

۱- ایجاد مشهای هندسی ساده مانند کره، مکعب، و...
۲- ایجاد مش از فایلهای سه بعدی
۳- ایجاد مش از روی نقاط تعریف شده

در DirectX اشیا سه بعدی در قالب مشهایی در صحنه رسم می شوند. یک Mesh از سه قسمت تشکیل شده است:

۱- نقاط و اضلاع (VertexBuffer و IndexBuffer)
۲- Materialها (آرایه ای از نوع Material)
۳- Textureها (آرایه ای از نوع Texture)

در واقع یک Mesh ساختاری است که تمام اطلاعات لازم برای رسم یک شی سه بعدی را در خود نگه می دارد. به عنوان مثال در فصل قبلی از یک Mesh جهت نگهداری VertexBuffer و IndexBuffer زمین و در کل، ایجاد یک شی یکپارچه برای زمین استفاده شد. علاوه بر آن Mesh دارای متدهای بسیار سودمندی مانند

ComputeNormalData و OptimizeInPlace بود که نحوه استفاده از آنها بررسی شد. علاوه بر اندیسهای نقاط، Texture و Matrial پوشاننده شی سه بعدی را نیز می توان در Mesh نگهداری کرد.

تمام مشهای ساده و سه بعدی از قسمتهای مختلفی تشکیل شده اند که به هر یک از این قسمتها در اصطلاح subset گفته می شود. مشهای ساده همیشه یک subset هستند. در مشهای سه بعدی نیز subset های مختلفی وجود دارد که هر subset دارای Texture و Matrial مخصوص به خود می باشد.

برای استفاده از مشهای ساده کافی است یکی از انواع زیر را انتخاب و داخل برنامه تعریف کرد.

Box	مکعب (device، عرض، ارتفاع، عمق)
Sphere	کره (device، شعاع، تعداد قطاعهای افقی، تعداد قطاعهای عمودی)
Cylinder	سیلندر (device، شعاع اول، شعاع دوم، طول، تعداد قطاعهای افقی، تعداد قطاعهای عمودی)
Polygon	چندضلعی (device، طول، تعداد وجه)
Teapot	قوری (device)
Torus	حلقه (device، شعاع داخلی، شعاع خارجی، تعداد وجه ها، تعداد حلقه ها)

برای مثال برای ایجاد Mesh کره می توان نوشت:

```
private Mesh mesh;
mesh = Mesh.Sphere(device, 30, 50, 50);
```

و در داخل متد OnPaint بین BeginScene و EndScene نیز نوشت:

```
mesh.DrawSubset(0);
```

برای ایجاد Mesh های دیگر نیز می توان نوشت:

```
mesh = Mesh.Box(device, 30, 50, 50);
mesh = Mesh.Cylinder(device, 30, 30, 50, 5, 8);
mesh = Mesh.Polygon(device, 30, 5);
mesh = Mesh.Teapot(device);
mesh = Mesh.Torus(device, 10, 30, 5, 7);
```

قالب کلی برای ایجاد Mesh از روی نقاط تعریف شده نیز به شرح ذیل است:

```
private Mesh mesh;
mesh = new Mesh (mesh تعداد مثلثهای, mesh تعداد نقاط, نوع حافظه, نوع نقاط, device);
mesh.SetVertexBufferData(نقاط);
mesh.SetIndexBufferData(اندیسها);
mesh.DrawSubset(0);
```

ذکر این نکته نیز ضروری است که تمام فایلهای سه بعدی دارای texture نیستند و بنابراین باید دقت شود که از فایلهای سه بعدی دارای texture در صحنه استفاده شود. علاوه براین در DirectX تمام مشهای ساده و سه بعدی ابتدا در مرکز جهان رسم می شوند و در صورت نیاز بایستی آنها را به محل مناسب انتقال داد. به عبارت دیگر قبل از ترسیم مشهای سه بعدی معمولا لازم است یک دوران، انتقال و تغییرمقیاس در جهان اعمال شود تا شی سه بعدی در محل مورد نیاز مستقر گردد.

در ادامه، شهر ساخته شده را به صورت یک Mesh یکپارچه در می آوریم. از آنجایی که این عمل را در بخش قبل انجام داده ایم، در این بخش انجام آن دوباره بررسی نمی شود.

می توان Mesh را در فایلی سه بعدی ذخیره کرده و دوباره آن را بارگذاری کرد. متد کلی LoadMesh، نام یک فایل را گرفته و Mesh داخل آن را در برنامه بارگذاری می کند:

```
private void LoadMesh (string Filename, ref Mesh mesh,
        ref Material[] MeshMaterials, ref Texture[] MeshTextures,
        ref float MeshRadius)
{ … }
```

در این متد، به جز نام فایل تمام پارامترهای دیگر دارای پیشوند ref می‌باشند. وجود این پیشوند قبل از پارامترها به این معنی است که مقدار آنها در داخل متد مشخص شده و باید در پارامترهای معادلشان کپی شده و به برنامه اصلی برگردانده شوند. در صورت عدم قرار دادن این پیشوند مقادیر داخل این پارامترها به صورت محلی فقط در داخل متد معتبر خواهند بود. در متد فوق علاوه بر Mesh پارامترهای دیگری مانند آرایه‌ای از Matrialها و آرایه‌ای از Textureها و یک عدد اعشاری برای نگهداری شعاع Mesh به متد ارسال شده و در داخل آن مقداردهی می‌شود. از این شعاع برای بررسی برخورد اشیا استفاده می شود.

دو دستور زیر را به بدنه متد LoadMesh اضافه می کنیم:

```
ExtendedMaterial[] MaterialArray;
mesh = Mesh.FromFile(Filename, MeshFlags.Managed, device,
                     out MaterialArray);
```

متد FromFile یک Mesh را از فایل مشخص شده، بارگذاری می کند. توسط پارامتر دوم مدیریت حافظه بر عهده VertexBuffer و IndexBuffer گذاشته می شود. پارامتر سوم device و پارامتر چهارم آرایه‌ای است که بقیه اطلاعات Mesh در آن قرار خواهد گرفت. پیشوند out نیز عملی تقریبا مشابه با ref دارد. پس از اجرای این دستورات، Mesh و تمامی اطلاعات جانبی مورد نیاز آن به برنامه وارد می شود.

لازم است ابتدا آرایه MaterialArray را بررسی کرده و در صورت خالی نبودن، اطلاعات داخل آن را پردازش کنیم:

```
if ((MaterialArray != null) && (MaterialArray.Length > 0))
{
  MeshMaterials = new Material[MaterialArray.Length];
  MeshTextures = new Texture[MaterialArray.Length];
  for (int i = 0; i < MaterialArray.Length; i++)
  {
    MeshMaterials[i] = MaterialArray[i].Material3D;
    MeshMaterials[i].Ambient = MeshMaterials[i].Diffuse;
    If ((MaterialArray[i].TextureFilename != null) &&
        (MaterialArray[i].TextureFilename != string.Empty))
    {
      MeshTextures[i] = TextureLoader.FromFile(device,
                                MaterialArray[i].TextureFilename);
    }
  }
}
```

در واقع هدف کد بالا، استخراج اطلاعات Material و Texture شی از آرایه اولیه به دو آرایه مجزا است. به ازای هر زیر شی در Mesh یک درایه در آرایه وجود دارد. ابتدا Material شی و سپس رنگ آن از آرایه استخراج می-شود. سپس وجود Texture برای Mesh بررسی شده و در صورت لزوم از فایل مربوطه به آرایه ای مجزا بارگذاری می شود.

تا این لحظه Mesh و آرایه های Material و Texture آن را از فایل استخراج کرده ایم. از آنجایی که احتمال دارد مقدار normal نقاط داخل Mesh به هر دلیلی محاسبه نشده باشد، به کمک کد زیر این کار را انجام می‌دهیم:

```
mesh = mesh.Clone(mesh.Options.Value,
                  CustomVertex.PositionNormalTextured.Format, device);
mesh.ComputeNormals();
```

در صورت استفاده از نور در محیط، Mesh کاملا طبیعی مشاهده خواهد شد.

برای یافتن شعاع Mesh بارگذاری شده، نقاط داخل Mesh را در قالب یک شی GraphicsStream از آن استخراج می کنیم. این شعاع برای بررسی برخورد دو شی استفاده می شود:

```
VertexBuffer vertices = mesh.VertexBuffer;
GraphicsStream stream = vertices.Lock(0, 0, LockFlags.None);
Vector3 MeshCenter;
MeshRadius = Geometry.ComputeBoundingSphere(stream,
                          mesh.NumberVertices, mesh.VertexFormat,
                          out MeshCenter)*Scaling;
vertices.Unlock();
```

کد بالا تمام محاسبات ریاضی لازم برای یافتن شعاع را انجام می دهد. مختصات مرکز Mesh نیز محاسبه می‌شود (هر چند که نیازی نیست). شعاع به دست آمده در مقدار Scailing محیط ضرب می شود. حال بایستی متغیرهای زیر را در کلاس تعریف کنیم:

```
private Mesh SpaceMesh;
private Material[] SpaceMeshMaterials;
private Texture[] SpaceMeshTextures;
private float SpaceMeshRadius;
private float Scaling = 0.005f;
```

اولین متغیر شی هواپیما و آخرین متغیر اندازه آن را نگهداری خواهد کرد. از آنجایی که بعدا ممکن است مشهای مختلفی به پروژه اضافه شود، متد جدیدی در برنامه تعریف می کنیم تا تمام بارگذاریها در آن انجام شود:

```
private void LoadMeshes()
{
  LoadMesh("mesh.x", ref SpaceMesh, ref SpaceMeshMaterials,
           ref SpaceMeshTextures, ref SpaceMeshRadius);
}
```

متد فوق را بایستی در تابع سازنده فراخوانی نمود. در ضمن از آنجایی که کلاس Mesh در فضای نام Microsoft.DirectX.Direct3DX تعریف شده است، بایستی این فضای نام هم به پروژه افزوده شود.

اگر همین الان برنامه نوشته شده را اجرا کنید تغییری مشاهده نخواهید کرد زیرا ما Mesh را در برنامه بارگذاری کرده ایم اما هنوز آن را نمایش نداده ایم. برای این منظور متد زیر را به کلاس اضافه می کنیم:

```
private void DrawMesh(Mesh mesh, Material[] MeshMaterials,
                      Texture[] MeshTextures)
{
  for (int i = 0; i < MeshMaterials.Length; i++)
  {
    device.Material = MeshMaterials[i];
    device.SetTexture(0, MeshTextures[i]);
    mesh.DrawSubset(i);
  }
}
```

به ازای تمام زیرشی‌های موجود در Mesh این حلقه یکبار تکرار شده و هر زیر شی با Materials و Texture مربوطه‌اش رسم می شود. متد فوق را داخل متد OnPaint فراخوانی می کنیم. البته قبل از فراخوانی این متد، محیط را تغییر مقیاس داده و اندکی دوران می دهیم:

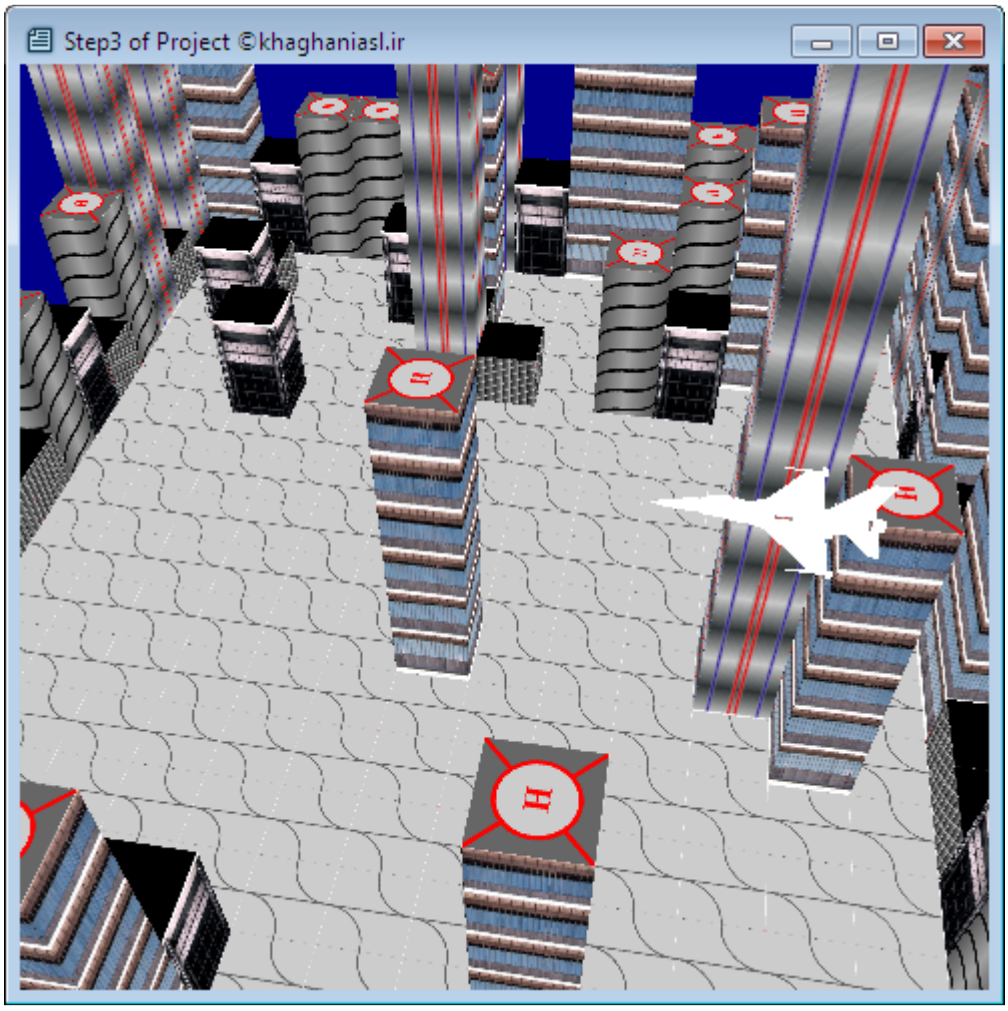
```
device.Transform.World = Matrix.Scaling(Scaling, Scaling, Scaling) *
Matrix.RotationX((float)Math.PI / 2) * Matrix.Translation(19, 5, 12);
DrawMesh(SpaceMesh, SpaceMeshMaterials, SpaceMeshTextures);
```

در کد فوق ابتدا هواپیما تغییر مقیاس داده می شود تا با اندازه شهر متناسب شود. در غیر این صورت هواپیما بسیار بزرگتر از شهر دیده خواهد شد. سپس به اندازه ۹۰ درجه حول محور x دوران داده می شود تا به صورت افقی بر روی شهر قرار گیرد و در نهایت به مکان مناسبی که داخل کادر دوربین باشد، منتقل می شود تا به خوبی دیده شود.

البته اگر برنامه را اجرا کنید، تنها هواپیما در صحنه مشاهده خواهد شد زیرا جهان، آخرین تغییرات اعمال شده بر روی خود را در صحنه های بعدی حفظ کرده و شهر نیز به همان نسبت با هواپیما کوچکتر خواهد شد و در نتیجه تقریبا محو می شود. برای حل این مشکل، هر بار برای رسم صحنه جدید ماتریس جهان را Reset می کنیم تا تغییرات قبلی در این ماتریس بر روی صحنه جدید اعمال نشود. برای این منظور کافی است کد زیر را قبل از رسم شهر در متد OnPaint درج کنیم:

```
device.Transform.World = Matrix.Identity;
```

اکنون با اجرای برنامه، صحنه ای مشابه زیر دیده خواهد شد:



۳-۳-۲ بهینه سازی Meshها

بیشتر فایلهای سه بعدی که توسط نرم افزارهای سه بعدی ساز تولید می شوند از تعداد نقاط و مثلثهای زیادی تشکیل می شوند که بارگذاری آنها روی صحنه به تعداد زیاد، باعث کاهش کارایی برنامه می شود. برای حل این مشکل می توان درون Mesh، نقاط نزدیک به یکدیگر را با هم ادغام کرده و در یک صحنه قرار داد.

برای کاهش تعداد نقاط یک Mesh، به آرایه ای از اعداد صحیح نیاز داریم تا اطلاعات فعلی نقاط را نگهداری کند. خطوط زیر باعث می شود تمام نقاطی که در Mesh فاصله آنها کمتر از ۰.۵ واحد باشد، با هم ادغام شوند:

```
int[] adj = new int[mesh.NumberFaces * 3];
mesh.GenerateAdjacency(0.5f, adj);
mesh.OptimizeInPlace(MeshFlags.OptimizeVertexCache, adj);
```

۳-۴ افزودن نور به محیط

از آنجایی که قبلا درباره مبانی نورپردازی به محیط بحث شده، مطالب این بخش به صورت خلاصه ارائه می شود. برای افزودن دو منبع نور Directional به محیط، ابتدا در برنامه قبلی نورپردازی محیط را فعال کرده و کدهای زیر را در ادامه اضافه می کنیم:

```
device.RenderState.Lighting = true;
device.Lights[0].Type = LightType.Directional;
device.Lights[0].Diffuse = Color.White;
device.Lights[0].Direction = new Vector3(1, 1, -1);
device.Lights[0].Update();
device.Lights[0].Enabled = true;
device.Lights[1].Type = LightType.Directional;
device.Lights[1].Diffuse = Color.White;
device.Lights[1].Direction = new Vector3(-1, -1, -1);
device.Lights[1].Update();
device.Lights[1].Enabled = true;
```

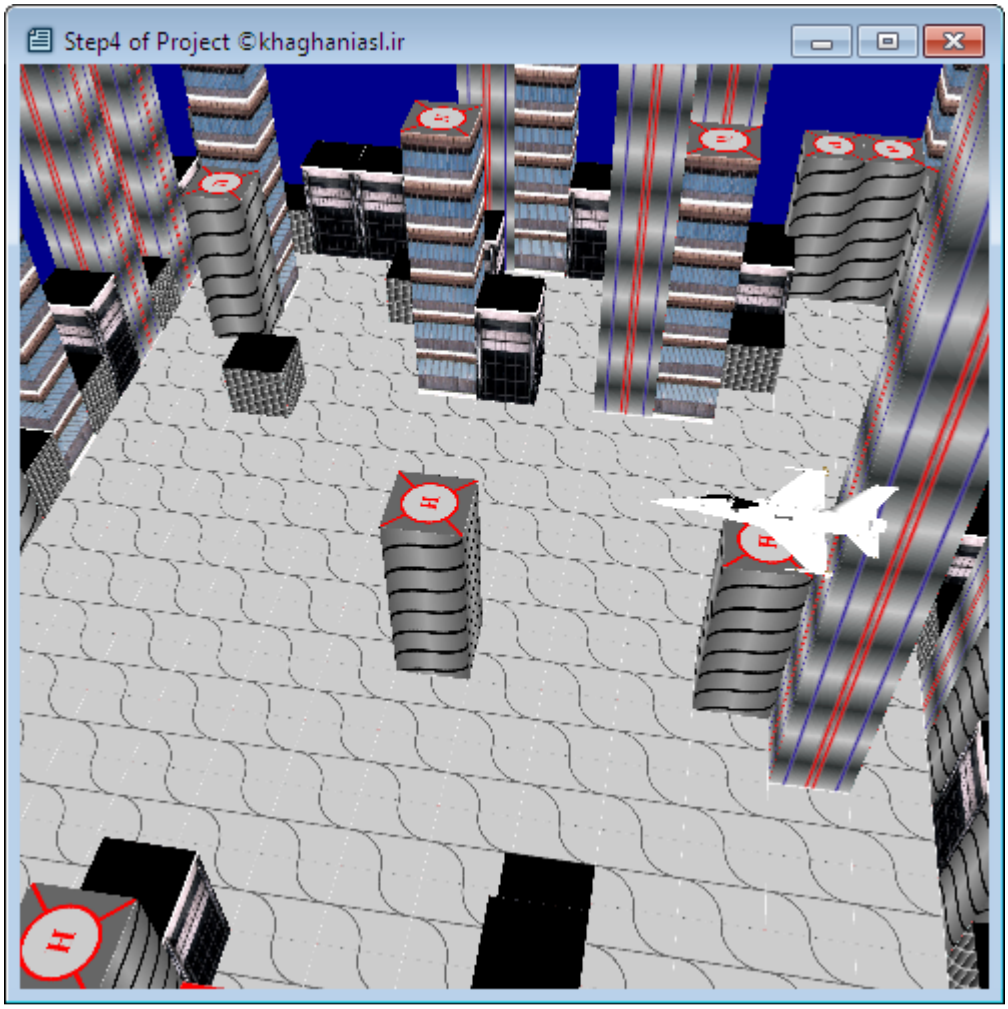
با اجرای برنامه، مشاهده خواهید کرد که فضای اطراف هواپیما روشن شده اما بقیه شهر تاریک مانده است. لذا برای روشن شدن تمام شهر نیاز به تعریف نورهای بیشتر در قسمتهای مختلف می باشد. البته به جای تعریف تک تک منابع نور بصورت مجزا، می توان از تکنیک دیگری استفاده کرد.

از آنجایی در مشها فقط مختصات و بردار نرمال نگهداری می شود و رنگ نگهداری نمی شود لذا زمانی که نورپردازی خاموش است تمام مشها سفید دیده می شوند و وقتی نورپردازی روشن است تمام مشها سیاه هستند. اگر منبع نوری تعریف نکرده باشیم برای اعمال رنگ می توانیم از نور محیطی استفاده کنیم.

نور محیطی نوعی نور که بعلت پخش زیاد، روی همه نقاط صحنه اثر یکسانی دارد و در واقع برای تنظیم روشنایی کلی محیط بکار می رود. ماده ها نیز می توانند بر حسب خصوصیت محیطی خود به این نور پاسخ دهند. کافی است دستور زیر را بین BeginScene و EndScene در متد OnPaint قرار می دهیم:

```
device.RenderState.Ambient = Color.DarkGray;
```

این دستور باعث توزیع نور به صورت یکنواخت در تمام فضای شهر خواهد شد و باعث می شود تمام اشکال رنگ بگیرند. البته تکنیکهای پیشرفته تری نیز وجود دارد که می تواند نورپردازی زیباتری در محیط ایجاد نماید.



۳-۵ حرکت هواپیما در محیط

ابتدا دو متغیر در برنامه تعریف می کنیم که موقعیت و زاویه حرکت هواپیما در شهر را نگهداری می کنند:

```
private Vector3 SpaceMeshPosition = new Vector3(8, 2, 1);
private Vector3 SpaceMeshAngles = new Vector3(0, 0, 0);
```

در این دو خط نقطه شروع و زاویه دوران تعریف و مقداردهی شده است. بدیهی است هواپیما باید در فضای شهر حرکت کرده و با زوایای دلخواه دوران داشته باشد اما پیاده سازی این کار واقعا عملی نیست. به جای حرکت دادن هواپیما در شهر، آن را همیشه در موقعیت (۰،۰،۰) رسم کرده و دوربین را پشت آن قرار می دهیم. به این ترتیب نیازی به حرکت دوربین همراه با حرکت هواپیما نخواهد بود. برای شبیه سازی حرکت هواپیما نیز، شهر را حرکت داده و یا دوران می دهیم.

برای رسم هواپیما در موقعیت (۰،۰،۰) دستورات زیر را در داخل متد OnPaint جایگزین دستورات قبلی می-کنیم:

```
device.Transform.World = Matrix.Scaling(Scaling, Scaling,
Scaling) * Matrix.RotationX((float)Math.PI / 2);
DrawMesh(SpaceMesh, SpaceMeshMaterials, SpaceMeshTextures);
```

دستور اول باعث کوچکتر شدن اندازه هواپیما و دوران آن حول محور Xها شده تا به صورت افقی در فضا قرار گیرد. دستور دوم نیز هواپیما را در مرکز صحنه رسم می نماید.

با تغییر متد Camera نیز دوربین را در پشت هواپیما مستقر می کنیم:

```
private void Camera()
{
  device.Transform.Projection = Matrix.PerspectiveFovLH(
(float)Math.PI/4,(float)this.Width/(float)this.Height,0.3f,500f);
  device.Transform.View = Matrix.LookAtLH(new Vector3(0, -1f,
0.2f), new Vector3(0, 0, 0), new Vector3(0, 0, 1));
}
```

از آنجایی که هواپیما در مرکز محورهای مختصات رسم شده، دوربین را اندکی عقبتر و بالاتر از آن مستقر کرده و محل نگرش آن را نیز مرکز محورهای مختصات قرار داده ایم. حداقل فاصله رویت اشیا در ClipingPlane را نیز ۰.۳ قرار داده ایم. اگر این مقدار ۱ باقی بماند، قسمت عقب هواپیما توسط دوربین رویت نخواهد شد.

حال نوبت رسم و حرکت دادن شهر است. برای این کار ابتدا دستور زیر را در متد OnPaint بعد از دستور توزیع نور بجای دستور device.Transform.World = Matrix.Identity جایگزین می کنیم:

```
device.Transform.World = Matrix.Translation(
-(float)SpaceMeshPosition.X, -(float)SpaceMeshPosition.Y,
-(float)SpaceMeshPosition.Z) *
Matrix.RotationYawPitchRoll((float)SpaceMeshAngles.X,
(float)SpaceMeshAngles.Y, (float)SpaceMeshAngles.Z);
```

متغیری جهت هماهنگی سرعت حرکت و سرعت سیستم نیز در کلاس تعریف می کنیم:

```
private float Speed = 0.04f;
```

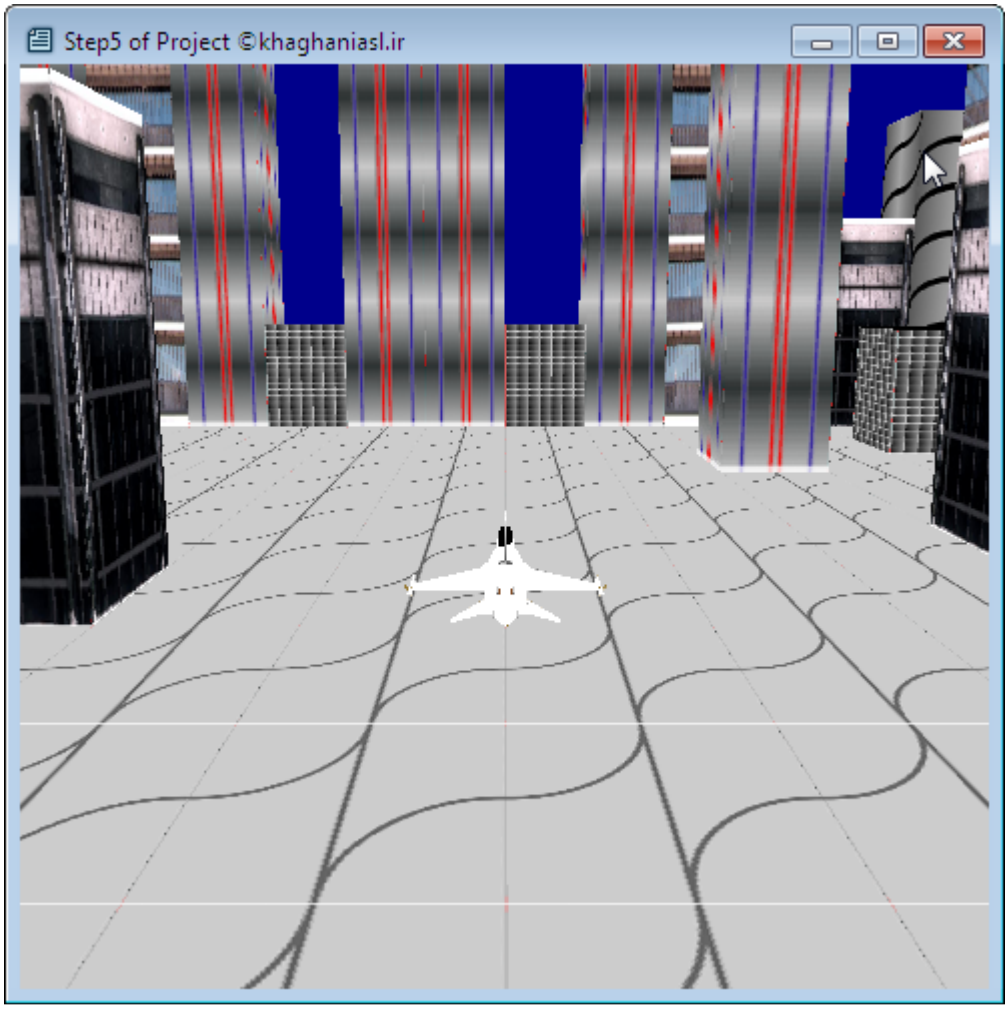
سپس متدی جدیدی به نام UpdatePosition به شکل زیر تعریف می کنیم که حرکت شهر را مطابق زاویه و سرعت مورد نظر شبیه سازی نماید:

```
private void UpdatePosition(ref Vector3 Position, Vector3 Angles,
float Speed)
{
  Vector3 AddVector = new Vector3();
  AddVector.X += (float)(Math.Sin(Angles.Z));
  AddVector.Y += (float)(Math.Cos(Angles.Z));
  AddVector.Z -= (float)(Math.Tan(Angles.Y));
  AddVector.Normalize();
  Position += AddVector * Speed;
}
```

حال کافی است متد UpdatePosition را در ابتدای متد OnPaint فراخوانی نمائیم:

```
UpdatePosition(ref SpaceMeshPosition, SpaceMeshAngles, Speed);
```

با اجرای برنامه، شاهد حرکت هواپیما در فضای شهر، یا به عبارت دقیقتر حرکت شهر نسبت به هواپیما خواهید بود:



۳-۶ کنترل هواپیما در محیط

در فصل قبل نحوه استفاده از صفحه کلید بیان شد. ابتدا فضای نام زیر را به رفرنسهای پروژه اضافه کرده و فراخوانی می کنیم:

```
using Microsoft.DirectX.DirectInput;
```

متغیری نیز در کلاس به شکل زیر تعریف می کنیم:

```
private Microsoft.DirectX.DirectInput.Device keyboard;
```

همان طور که قبلا گفته شد، متغیر keyboard جهت ارتباط با صفحه کلید است. همانند قبل، متدی به نام InitializeKeyboard به شکل زیر جهت پیکربندی صفحه کلید تعریف می نمائیم:

```
public void InitializeKeyboard()
{
  keyboard = new Microsoft.DirectX.DirectInput.Device
             (SystemGuid.Keyboard);
  keyboard.SetCooperativeLevel(this,CooperativeLevelFlags.Background|
                               CooperativeLevelFlags.NonExclusive);
  keyboard.Acquire();
}
```

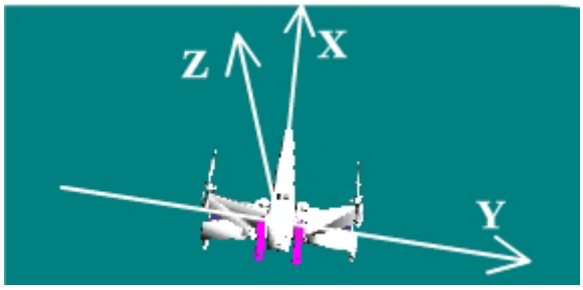
لازم به یادآوری است که متد فوق بایستی در تابع سازنده فرم فراخوانی شود:

```
public Form1()
{
  InitializeComponent();
  …
  InitializeKeyboard();
}
```

حال متدی به نام ReadKeyboard تعریف می کنیم تا به ازای فشردن کلیدهای جهتی چپ و راست، به زاویه چرخش هواپیما 2.5f اضافه یا کاسته شود. این متد بایستی در متد OnPaint فراخوانی گردد.

```
private void ReadKeyboard()
{
  KeyboardState keys = keyboard.GetCurrentKeyboardState();
  if (keys[Key.RightArrow])
  {
    SpaceMeshAngles.X -= 2.5f * Speed;
  }
  if (keys[Key.LeftArrow])
  {
    SpaceMeshAngles.X += 2.5f * Speed;
  }
}
```

در حقیقت زوایای چرخش روی هواپیما به شکل زیر است:

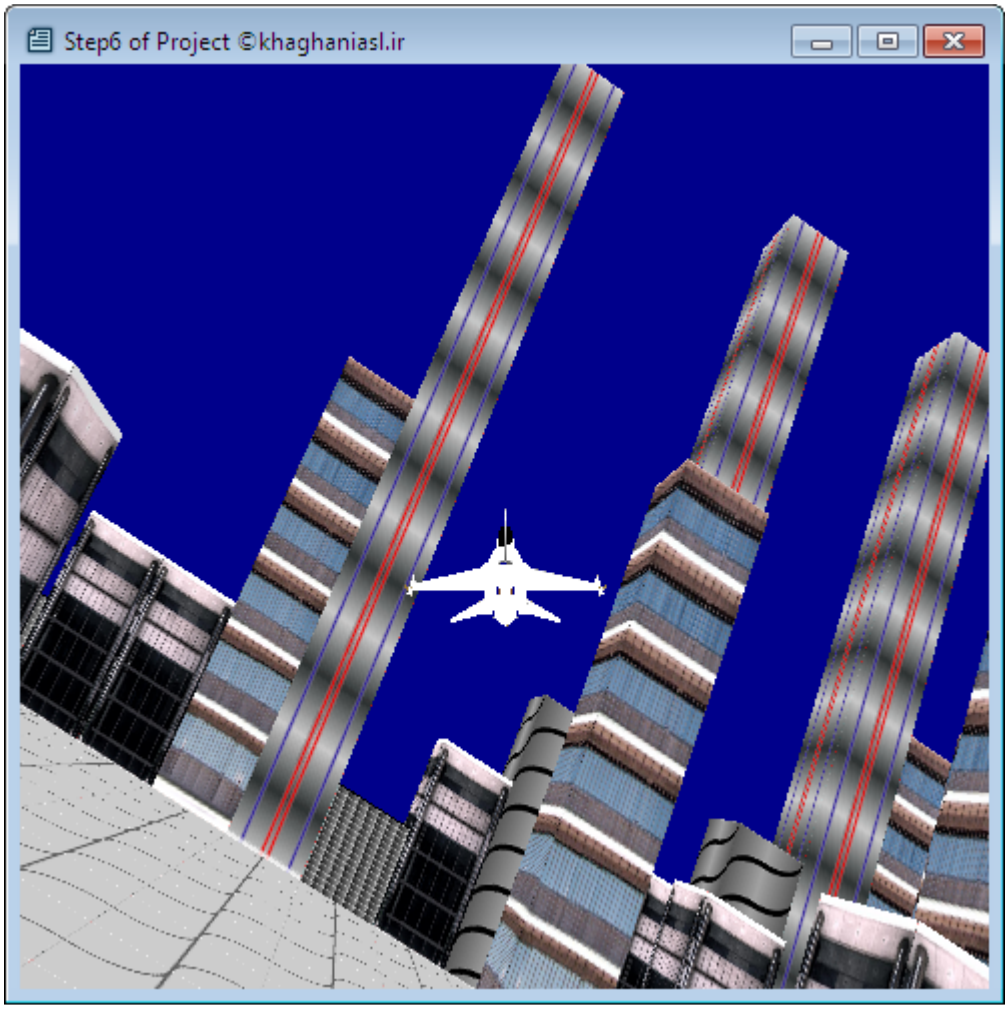


به ازای فشردن کلیدهای جهتی چپ و راست، جهت حرکت هواپیما (در حقیقت زوایای رسم شهر به کمک متد UpdatePosition) تغییر خواهد کرد. اگر روی محور xها باشد، هواپیما حول خودش به چپ و راست خواهد چرخید. اگر محور yها را انتخاب کنیم، هواپیما به طرف بالا و پائین (مشابه صعود و فرود) حرکت خواهد کرد و اگر محور zها انتخاب شود، هواپیما بدون هیچ چرخشی به چپ و راست حرکت خواهد کرد.

با افزودن دستورات زیر به متد ReadKeyboard، صعود و فرود هواپیما نیز به وسیله کلیدهای جهتی بالا و پائین، قابل کنترل خواهد بود:

```
if (keys[Key.DownArrow])
{
  SpaceMeshAngles.Z += (float)(Speed *
                               Math.Sin(SpaceMeshAngles.X));
  SpaceMeshAngles.Y += (float)(Speed *
                               Math.Cos(SpaceMeshAngles.X));
}
if (keys[Key.UpArrow])
{
  SpaceMeshAngles.Z -= (float)(Speed *
                               Math.Sin(SpaceMeshAngles.X));
  SpaceMeshAngles.Y -= (float)(Speed *
                               Math.Cos(SpaceMeshAngles.X));
}
```

حال با اجرای برنامه، شما قادر به حرکت دادن هواپیما در فضای سه بعدی شهر خواهید بود:



۳-۷ تشخیص برخورد

برای تشخیص برخورد هواپیما با یک عنصر از صحنه سه بعدی، قصد داریم هواپیما را به عنوان یک کره مدل نمائیم که هنگام بارگذاری Mesh، شعاع کره در متغیری به نام SpaceMeshRadius ذخیره شود.

با استفاده از برخی از اعمال ساده ریاضی متدی به نام CheckCollision تعریف می کنیم که یک عدد صحیح باز خواهد گرداند. عدد ۰ به معنی عدم برخورد و عدد ۱ به معنی برخورد هواپیما با یک شی از صحنه و عدد ۲ به معنی بیش از حد بالا رفتن هواپیما خواهد بود. البته این متد نیاز به موقعیت هواپیما و شعاع کره دارد:

```
private int CheckCollision(Vector3 position, float radius)
{
  if (position.Z - radius < 0) return 1;
  if (position.Z + radius > 15) return 2;
  if ((position.X < -10) || (position.X > Width + 10)) return 2;
  if ((position.Y < -10) || (position.Y > Height + 10)) return 2;
  if ((position.X - radius > 0) && (position.X + radius < Width)
  && (position.Y - radius > 0) && (position.Y + radius < Height))
  {
     if (position.Z - radius < BuildingHeights[FloorPlan[
                    (int)position.X, (int)position.Y]]) return 1;
  }
  return 0;
}
```

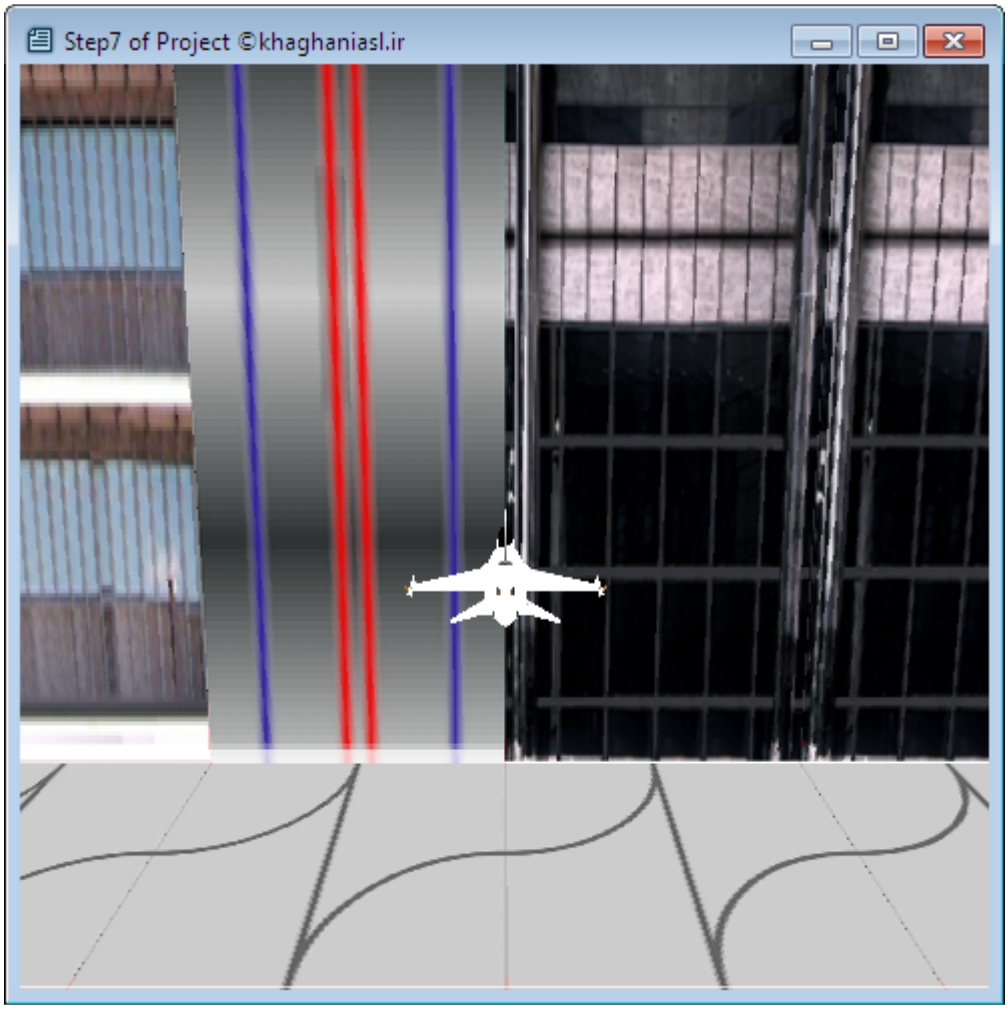
در خط اول سقوط هواپیما بررسی می شود. position.Z متناظر با مرکز هواپیما است و شعاع را از آن کم می-کنیم تا پایین ترین نقطه هواپیما بدست آید. اگر این نقطه کمتر از صفر باشد، هواپیما سقوط کرده است و تابع مقدار ۱ را برمی‌گرداند. در خط دوم نیز اگر هواپیما بیش از حد بالا رفته است، تابع مقدار ۲ را برمی‌گرداند. به این ترتیب می توان نوع برخورد را تشخیص داد.

دو خط بعدی مشابه همین بررسی است و اگر هواپیما بیش از حد از شهر دور شده باشد، تابع مقدار ۲ را برمی-گرداند. خط پنجم برای بررسی برخورد هواپیما با ساختمانها است. این دستورات بررسی می کند که در آن موقعیت، آیا پایین ترین نقطه هواپیما پایین تر از بالای ساختمان است؟ به یاد داشته باشید که FloorPlan شامل نوع ساختمان در آن موقعیت است و ما می توانیم ارتفاع متناظر را در آرایه BuildingHeights بیابیم. واضح است که با توجه به FloorPlan، مختصات x و y هواپیما نباید کمتر از صفر یا بزرگتر از ابعاد این آرایه باشد. در انتها، اگر هیچ یک از شرطها مثبت نباشد، تابع مقدار صفر را برمی‌گرداند که به معنی عدم برخورد خواهد بود.

حال تنها کار باقیمانده، واکنش به برخورد است. پس خطوط زیر را به متد OnPaint اضافه می کنیم تا در صورت برخورد هواپیما با ساختمانها، سقوط هواپیما یا اوج گرفتن بیش از حد آن، موقعیت هواپیما به مکان اولیه برگردد:

```
if (CheckCollision(SpaceMeshPosition, SpaceMeshRadius) > 0)
{
   SpaceMeshPosition = new Vector3(8, 2, 1);
   SpaceMeshAngles = new Vector3(0, 0, 0);
}
```

حتی می توان با هر بار برخورد، سرعت حرکت هواپیما را کاهش داد و یا متغیری در نظر گرفت که تعداد برخوردها را ثبت نماید.



۳-۸ ترسیم SkyBox

تاکنون رنگ پس زمینه اطراف شهر ثابت و تک رنگ بوده ولی می توان برای محیط از SkyBox استفاده کرد. یک SkyBox چیزی بیش از یک مش نیست و کافی است یک بار این مش بارگذاری شود تا بتوان بطور خودکار از بافتهای موجود آن استفاده کرد. برای این کار یک متغیر از نوع Mesh جهت SkyBox و دو آرایه جهت ذخیره سازی متریالها و بافتهای آن در برنامه تعریف می کنیم:

```
private Mesh SkyBoxMesh;
private Material[] SkyBoxMaterials;
private Texture[] SkyBoxTextures;
```

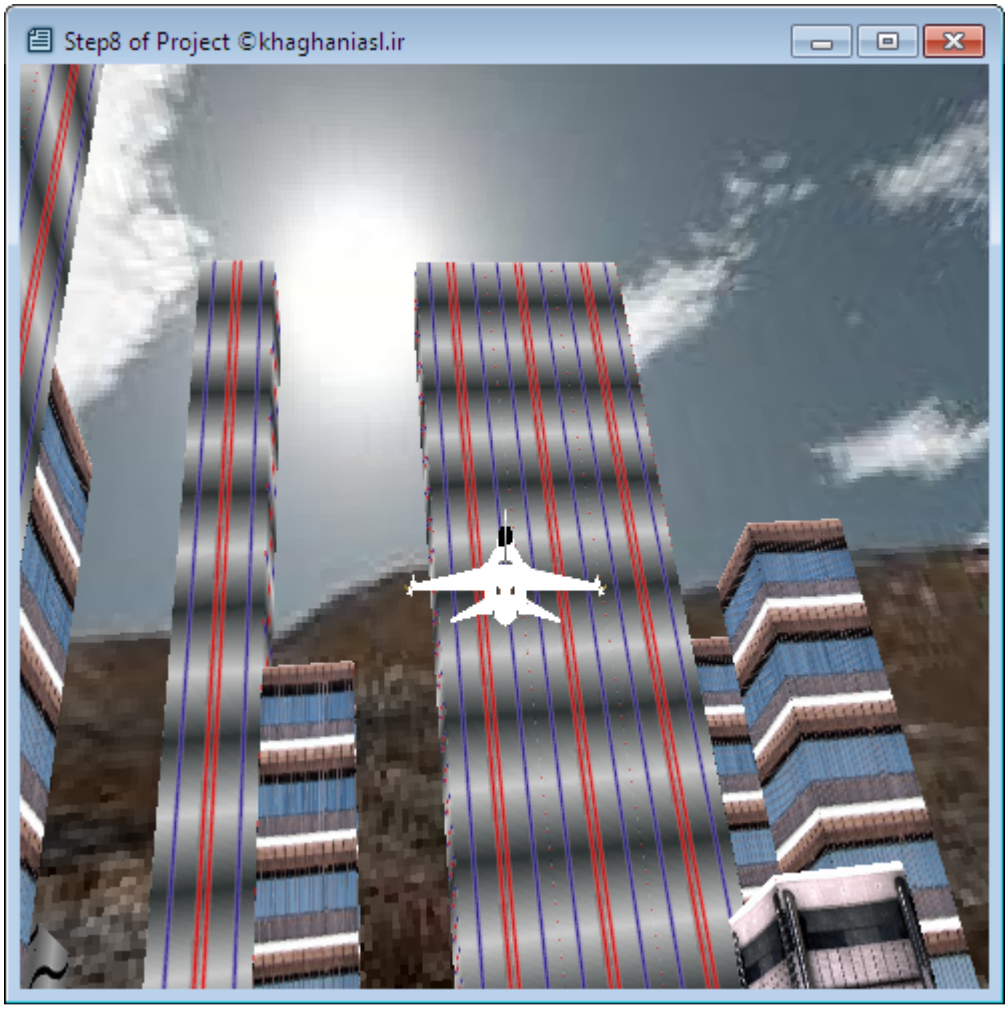
حال کافی است دستورات زیر را جهت بارگذاری Mesh مربوط به SkyBox به متد LoadMeshes اضافه کنیم:

```
private void LoadMeshes()
{
  …
  float dummy = 0;
  LoadMesh("skybox.x", ref SkyBoxMesh, ref SkyBoxMaterials,
                                    ref SkyBoxTextures, ref dummy);
}
```

برای رسم SkyBox نیز دستورات زیر را به متد OnPaint اضافه می کنیم:

```
protected override void OnPaint(PaintEventArgs e)
{
   …
   device.RenderState.Ambient = Color.White;
   device.Transform.World = Matrix.RotationX((float)Math.PI / 2)*
            Matrix.RotationYawPitchRoll((float)SpaceMeshAngles.X,
             (float)SpaceMeshAngles.Y, (float)SpaceMeshAngles.Z);
   DrawMesh(SkyBoxMesh, SkyBoxMaterials, SkyBoxTextures);
   device.EndScene();
   …
}
```

همان طور که مشاهده می کنید نیازی به انتقال SkyBox نیست. کافی است SkyBox متناسب با چرخش هواپیما بچرخد. از طرف دیگر SkyBox نیازی به سایه هم ندارد. می توان با قرار دادن بافتهای آسمان و کوه در مسیر برنامه، برنامه را اجرا کرد تا اطراف شهر طبیعی تر به نظر برسد.



اگر در صحنه، مرز بین textureها قابل تشخیص بود، برای اینکه texture ها به آرامی در کنار هم ادغام شوند، می توان TextureFiltering را فعال کرد. مخصوصا زمانی که یک تصویر کوچک روی یک سطح بزرگ کشیده شود، پیکسلهای تصویر بزرگ شده و texture به حالت مشبک دیده می شود.

اعمال تکنیک TextureFiltering　　عدم اعمال تکنیک TextureFiltering

به صورت کلی از تکنیک TextureFiltering در دو موقعیت می توان استفاده کرد:

۱- برای شفاف نمودن تصاویری که بزرگتر شده اند.

۲- برای آمیختن رنگ پیکسلهای همسایه در یکدیگر.

کافی است دستورات زیر را به متد InitializeDevice برنامه اضافه کنیم:

```
device.SamplerState[0].MinFilter = TextureFilter.Anisotropic;
device.SamplerState[0].MagFilter = TextureFilter.Anisotropic;
device.SamplerState[0].AddressU = TextureAddress.Mirror;
device.SamplerState[0].AddressV = TextureAddress.Mirror;
```

۳-۹ افزودن صدا

برای اضافه کردن صدا به یک محیط سه بعدی، یک روش اضافه کردن رفرنس زیر به پروژه است:

```
using Microsoft.DirectX.DirectSound;
```

علاوه بر این، به یک device جهت ارتباط با کارت صوتی سیستم و یک بافر جهت نگهداری داده های بارگذاری شده از یک فایل صوتی نیاز داریم. پس دو متغیر زیر را تعریف می کنیم:

```
private Microsoft.DirectX.DirectSound.Device SoundDevice;
private SecondaryBuffer SoundBuffer;
```

متدی به نام InitializeSound تعریف می نمایم:

```
private void InitializeSound()
{
  SoundDevice = new Microsoft.DirectX.DirectSound.Device();
  SoundDevice.SetCooperativeLevel(this, CooperativeLevel.Normal);

  BufferDescription description = new BufferDescription();
  description.ControlEffects = false;
  SoundBuffer = new SecondaryBuffer("sound.wav", description,
                                    SoundDevice);
  SoundBuffer.Play(0, BufferPlayFlags.Default);
}
```

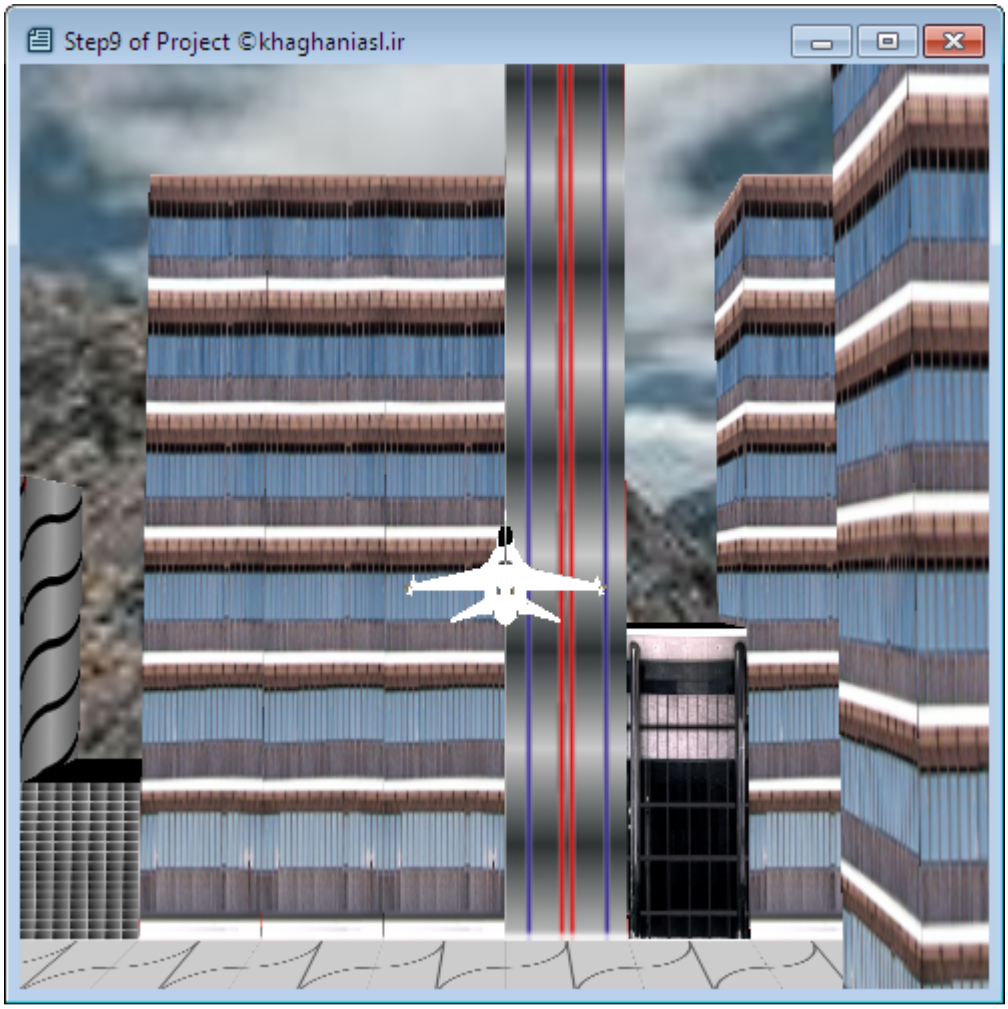
در متد فوق، ابتدا device مرتبط با کارت صوتی را مقداردهی اولیه شده است. تنظیمات CooperativeLevel را به شکل عادی قرار داده ایم تا مدیریت منابع آسان باشد. برای توصیف بافر، متغیر description را تعریف کرده و چون نیازی به کنترل صدا نداریم، خصوصیت ControlVolume آن را تعریف نکردیم.

می‌خواهیم در SecondaryBuffer از روش Clone استفاده نمائیم، بنابراین در خط بعدی ControlEffects را false انتخاب می‌کنیم. در خط پنجم متغیری به عنوان بافر تعریف کرده و فایل صوتی، توصیفات بافر و device مربوطه را به آن انتساب می دهیم. در خطر آخر نیز فایل صوتی بارگذاری شده پخش می شود. پارامتر اول نشان دهنده اولویت است و پارامتر دوم نحوه پخش را مشخص می نماید. به صورت پیش فرض فایل صوتی بارگذاری شده یکبار پخش خواهد شد. چنانچه تمایل داشته باشید فایل صوتی تکرار شود کافی است پارامتر دوم را BufferPlayFlags.Looping قرار دهید.

حال کافی است متد InitializeSound در تابع سازنده فرم فراخوانی شود:

```
public Form1()
{
   …
   InitializeSound();
}
```

با اجرای برنامه، فایل صوتی انتخاب شده پخش خواهد شد.



برای قرار دادن صدا به عنوان آهنگ زمینه می توان به روش ساده تری نیز عمل کرد. کافی است رفرنس Microsoft.DirectX.AudioVideoPlayback را به پروژه اضافه کرده و یک متغیر از نوع Audio تعریف کرده و در متد InitializeSound کدهای زیر را قرار دهیم:

```
private Audio BackgroudMusic;

private void InitializeSound()
{
  BackgroudMusic = new Audio("sound.mp3");
  BackgroudMusic.Play();
}
```

۳-۱۰ افزودن متن

برای نمایش متنی دلخواه در صحنه، کافی است متغیری به شکل زیر در برنامه تعریف کنیم:

```
private Microsoft.DirectX.Direct3D.Font SampleText;
```

متدی به نام InitializeFont جهت مقداردهی اولیه فونت متن تعریف کرده و در تابع سازنده فرم فراخوانی می-کنیم:

```
private void InitializeFont()
{
  System.Drawing.Font systemfont = new
              System.Drawing.Font("Tahoma", 14f, FontStyle.Bold);
```

```
  SampleText = new Microsoft.DirectX.Direct3D.Font(device,
                                                   systemfont);
}
```

حال کافی است دستور زیر را در متد OnPaint قرار دهیم:

```
SampleText.DrawText(null, string.Format("Welcome!", ""),
                                    new Point(10, 20), Color.White);
```

پارامتر دوم متن نمایشی، پارامتر سوم مختصاتی که نوشته در آنجا نمایش داده می شود و پارامتر چهارم رنگ متن نمایشی را مشخص می نماید. با اجرای برنامه، متن مورد نظر با رنگ انتخابی و فونت مشخص؛ در مختصات تعیین شده نمایش داده خواهد شد: