تشریح عملکرد الگوریتم معروف A* در مسیر یابی

(بکار بردن نمونه ساده ای از این الگوریتم در برنامه نویسی GML)

سطح کلی مقاله : پیشرفته

بهترین اندازه دید : 1024 * 768

طراحی و تنظیم : علی کسایی

WWW.Persian-Designers.COM

Ali_nwdo@yahoo.com

قبل از آغاز مطالعه این مقاله به خاطر داشته باشید که کدهای فراهم شده در این مقاله در حالت کلی برای زبان داخلی نرم افزار Game Maker یعنی زبان GML نوشته شده اند. بنابراین استفاده مستقیم از آنها در زبانهای دیگر نظیر C++ یا دلفی امکان پذیر نیست. اما با درک نحوه عملکرد این الگوریتم و با اندکی تجربه در برنامه نویسی خواهید توانست تا کدهای نچندان پیچیده فراهم شده را به زبان های دیگر نیز تبدیل کنید...

در انتها فرض بر این گذاشته میشود که خواننده در ریاضیات دارای پایه ای قوی بوده و تا حد لزوم از معادلات ریاضی و دروس مربوط به جبر خطی و هندسه تحلیلی آگاه است. داشتن اندکی اطلاعات در زمینه های ذکر شده به درک بهتر و بهینه تر این مقاله کمک بسیار خواهد نمود.

الگوریتم مسیر یابی *A (بخوانید A Star) یکی از معروفترین و قدرتمند ترین الگوریتمها برای یافتن کوتاهترین فاصله بین دو نقطه A و B در کوتاهترین زمان ممکن می باشد. این الگوریتم در بسیاری از بازیهای معتبر و معروف تجاری که نیازمند مسیریابی با دقت و سرعت بالا هستند استفاده شده است. به عنوان مثال بازیهای استراتژیک بیشترین استفاده را از الگوریتم فوق به عمل می آورند.

مشکلی که در این میان به چشم میخورد اینست که کاربران نرم افزار Game Maker بطور مستقیم قادر نیستند تا از این الگوریتم استفاده کنند. چرا که این کد به زبان GML تبدیل نشده است. البته تعجبی ندارد ، چرا که الگوریتم فوق بسیار گسترده است. در عین حال ساختار این الگوریتم برای بسیار از کاربران تازه کار بسیار مبهم و نا مفهوم بنظر میرسد. با این وجود این مقاله میکوشد تا بطور خلاصه مکانیزم عملکرد این الگوریتم را تشریح کند و دلایل استفاده و چگونگی کارکرد آنرا توضیح بدهد.

بررسی اولیه :

همان گونه که قبلا اشاره شد ، الگوریتم *A برای یافتن کوتاهترین فاصله بین دو نقطه بکار میرود. اما قبل از وارد شدن به جزییات ابتدا چند مفهوم ابتدایی و اساسی را از نظر میگذرانیم :

: Node

یک گره... که میتواند بیانگر یک مکان بر روی صفحه باشد. در این مقاله ما از گره به عنوان مختصاتی برای نقاط خاص بر روی شبکه های دو بعدی استفاده خواهیم کرد.

: List

یک مجموعه از اشیاء... آنها میتوانند عناصر مختلف ، اعداد ، حروف یا هر چیز دیگری باشند...

: Priority Queue

یک نوع خاص از لیستها... در واقع به نوعی از لیست گفته میشود که در آن عناصر موجود در لیست نسبت به یکدیگر حق تقدم دارند و میبایست این تقدم و تاخر در صف های مجازی رعایت شوند.

: Heap

یک حالت اجرایی از Priority Queue که در بالا معرفی شد.

: Recursion

یک تکنیک در برنامه نویسی... در واقع در زبان تخصصی به هنگامی گفته میشود که یک تابع خودش را فراخوانی کند...

: Heuristic

معادله یا رابطه ای که عنصر مورد نظر را از وضعیت اطرافش آگاه میکند...

البته همان گونه که خودتان میدانید ، مقالات و توضیحات بسیار کامل تر و بهتری در مورد این اصطلاحات در وب یافت میشوند. بنابراین چنانچه احساس میکنید که هنوز موضوع برایتان بطور مبهم باقی مانده است ، کافی است تا چرخی در وب زده و چند مقاله در مورد این مسایل مطالعه کنید. در ادامه این مقاله فرض بر این گذاشته شده است که خواننده با این اصطلاحات اولیه آشنا است.

بسیار خوب. به تشریح عملکرد الگوریتم بر میگردیم. کاری که این الگوریتم انجام میدهد آنست که در نقطه ابتدایی مشخص شده شروع به تولید Node میکند. سپس در Node های ایجاد شده بدنبال بهترین و مناسب ترین ها میگردد و هنگامی که مناسبترین نقطه را پیدا کرد آنرا به عنوان نقطه ابتدایی بعدی قرار میدهد و سپس مجددا به تولید Node های جدید در اطراف این نقطه جدید مشغول میشود. در عین حال نقاط نامرغوب (نقاطی که انتخاب نشده اند) نیز نادیده گرفته میشوند. به همین ترتیب الگوریتم به ایجاد نقاط و گره ها میپردازد تا سر انجام به نقطه انتهایی برسد و در این هنگام میسر متشکل از نقاط مرغوب به عنوان بهترین مسیر انتخاب میشود و توسط برنامه برگشت داده میشود.

سوالی که ایجاد میشود این است که دلیل انتخاب یک نقطه به عنوان نقطه مرغوب یا نقطه نامرغوب چیست ؟ در اینجا به مفهوم Heuristic میرسیم... به طور معمول یک Heuristic برای الگوریتم *A بصورت

F = G + H

می باشد که در آن G تعداد گامهایی است که برای رسیدن به گره فعلی برداشته شده است. H تخمینی است از تعداد گامهایی که برای رسیدن به گره هدف میبایست برداشته شوند و F نیز برابر با مجموع این دو است. با در نظر داشتن این نکته میتوان ارزش هر گره نسبت به سایر گره های موجود در اطراف آنرا مشخص نمود.

ساختارهای داده :

چنانچه مقاله را بدقت مطالعه کرده باشید به این نکته رسیده اید که ما اصلا این لیست های فرضی و Heap های اجرایی را نداریم تا بتوانیم بگونه ای باعث کارکردن این الگوریتم شویم. با این حال بخاطر داشته باشید تا همواره گره های مرغوب را انتخاب کنید و همچنین همواره گره های نامرغوب را کنار بگذارید. در عین حال ما نیازمندیم تا وضعیت **تمامی** نقاطی که پردازش کرده ایم و قرار است که پردازش کنیم را جمع آموری کنیم. البته در این بین شما آزادید تا برای این منظور از ساختارهای داده ، لیست داده و لیست اجرای مورد علاقه خود استفاده کنید. در واقع تحت یک عنصر تنها یک آرایه وجود دارد ، اما با این وجود ما از آن مانند یک آرایه عادی استفاده نمیکنیم. در زیر کدهایی آورده شده اند که نحوه تعریف و استفاده از این آرایه ها را نشان میدهند. این آرایه ها مانند Heap ها و List ها عمل میکنند. به دقت به کدها نگاه کنید و ببینید که آیا قادر به درک عملکرد آنها هستید ؟

در مورد این موضوع بخصوص دقت کنید که چگونه Heap ها ایجاد و اجرا شده اند. البته که آشکار نیست که آنها برای چه و چگونه کار میکنند ، اما فعلا این نکته را در ذهن داشته باشید که تنها راه خارج کردن بهینه یک داده از یک Heap ، خارج کردن مناسبترین گره با بالاترین Priority میباشد. در مورد مثال ما نقطه ای که دارای کمترین مقدار F باشد و ما این نقطه را Min Heap یا گروه مینیمم اطلاق خواهیم کرد.

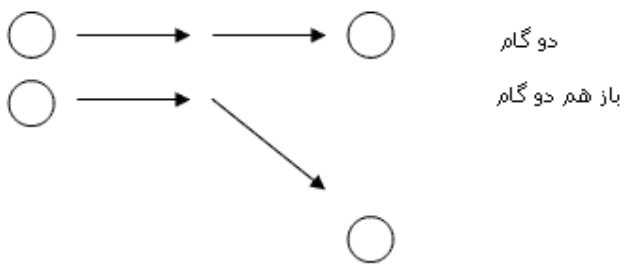
همچنین نگاهی به ساختارهای داده ای dataNode بیاندازید. اینها مجموعه گره هایی هستند که مورد محاسبه قرار خواهند گرفت. بنابراین از درک صحیح علت تعریف متغیر های تعریف شده برای هر گره منفرد ، اطمینان حاصل کنید.

Heuristic :

در مثال ما مفهوم Heuristic دارای ارزش ویژه ای میباشد. البته توجه داشته باشید که heuristic بکار رفته در الگوریتم اصلی *A بسیار پیچیده است ولی خوشبختانه در این مقاله ما از مفهوم های شبیه سازی شده ساده تری استفاده خواهیم کرد. به خاطر داشته باشید که هدف ما تخمین زدن این است که چه مقدار با گره مقصد فاصله داریم... با این وجود الگوریتم *A بهترین جواب را هنگامی به شما خواهد داد که از تخمین های کوچک استفاده کنید. چرا که استفاده از تخمین های بزرگ در موارد استثنایی توصیه میشود. فرض کنید که میخواهیم یک توپ کوچک را بطور مورب بسوی هدف هدایت کنیم. بنابراین میدانیم که کمترین مقدار حرکت در راستای رسیدن به هدف از رابطه ریاضی زیر بدست می آید :

```
max(abs(dx), abs(dy))
```

که در آن dx و dy تفاوت های موجود در مختصات x و y میباشند. بسیار خوب... این کد چگونه کار میکند ؟ شرایط ممکن را در نظر میگیریم :



همان طور که ملاحظه میکنید خیلی هم پیچیده به نظر نمیرسد. خوشبختانه این heuristic ساده تمام چیزی است که برای یک مسیر یابی قطری به آن نیاز داریم.

الگوریتم اصلی :

ساختار اصلی الگوریتم *A به صورت زیر است :

```
{
  while (dataHeap.numObj > 0) {
    n = HeapGet(dataHeap);
    if (n.x == objGoal.x && n.y == objGoal.y)
      return n;
    arrPossDirectionsX[1] = n.x + global.SNAPDIS;
arrPossDirectionsY[1] = n.y;
    arrPossDirectionsX[2] = n.x + global.SNAPDIS;
arrPossDirectionsY[2] = n.y - global.SNAPDIS;
    arrPossDirectionsX[3] = n.x;
arrPossDirectionsY[3] = n.y - global.SNAPDIS;
    arrPossDirectionsX[4] = n.x - global.SNAPDIS;
arrPossDirectionsY[4] = n.y - global.SNAPDIS;
    arrPossDirectionsX[5] = n.x - global.SNAPDIS;
arrPossDirectionsY[5] = n.y;
    arrPossDirectionsX[6] = n.x - global.SNAPDIS;
arrPossDirectionsY[6] = n.y + global.SNAPDIS;
    arrPossDirectionsX[7] = n.x;
arrPossDirectionsY[7] = n.y + global.SNAPDIS;
    arrPossDirectionsX[8] = n.x + global.SNAPDIS;
arrPossDirectionsY[8] = n.y + global.SNAPDIS;
    for (ii = 1; ii <= 8; ii += 1) {
      if (!place_meeting(arrPossDirectionsX[ii],
arrPossDirectionsY[ii], objWall)) {
        newNode = instance_create(arrPossDirectionsX[ii],
arrPossDirectionsY[ii], dataNode);
        newNode.move = ii;
        newg = n.g + 1;

      if (ListContains(dataList, newNode)) {
          if (ListFind(dataList, newNode).g <= newg)
             with (newNode) instance_destroy();
          else {
            newNode.parent = n;
            newNode.g = newg;
            newNode.h = Heuristic(newNode);
            newNode.f = newNode.g + newNode.h;
            if (ListContains(dataList, newNode)) ListRemove(dataList,
newNode);
            if (!ListContains(dataHeap, newNode)) HeapAdd(dataHeap,
newNode);
          }
        } else
        if (ListContains(dataHeap, newNode)) {
          if (ListFind(dataHeap, newNode).g <= newg)
            with (newNode) instance_destroy();
          else {
            newNode.parent = n;
            newNode.g = newg;
```

```
            newNode.h = Heuristic(newNode);
            newNode.f = newNode.g + newNode.h;
            if (ListContains(dataList, newNode)) ListRemove(dataList,
newNode);
            if (!ListContains(dataHeap, newNode)) HeapAdd(dataHeap,
newNode);
          }
        } else {
          newNode.parent = n;
          newNode.g = newg;
          newNode.h = Heuristic(newNode);
          newNode.f = newNode.g + newNode.h;
          if (ListContains(dataList, newNode)) ListRemove(dataList,
newNode);
          if (!ListContains(dataHeap, newNode)) HeapAdd(dataHeap,
newNode);
        }
      }
    }
    ListAdd(dataList, n);
  }
  return -1;
}
```

البته ممکن است این کد اندکی ترسناک بنظر برسد ولی چنانچه به خواندن مقاله ادامه دهید درک آن برایتان آسان خواهد شد.

ابتدا ما اولین گره را در شیء objRun ایجاد میکنیم. سپس نوع ساختار داده مورد نیازمان را ایجاد کرده و در نهایت :

```
while (dataHeap.numObj > 0) {
  n = HeapGet(dataHeap);
  if (n.x == objGoal.x && n.y == objGoal.y)
    return n;
```

به کد بالا میرسیم. ملاحظه میشود که در هر بار اجرای حلقه ما چیزهایی را از Heap استخراج میکنیم... چرا Heap ? زیرا همان طور که قبلا گفته شد هدف ما بدست آوردن گرهی با کوچکترین مقدار F است. گره با کوچکترین مقدار F به معنای بهترین گره برای مسیر یابی میباشد. به بیان دیگر این گره از سایر گره ها به گره مقصد نزدیکتر است. این دلیل جستجوی ما در تابع heap بدنبال آن است. در عین حال ما همواره چک میکنیم که آیا به مقصد رسیده ایم یا خیر...در صورت رسیدن از ادامه اجرای تابع خارج میشویم و کد ، مقدار n بدست آمده را برای ما باز میگرداند. توجه داشته باشید که ساختار تابع heap بکار رفته در کد بالا بصورت زیر میباشد که یک آرگومان را پذیرفته و عملیات محاسباتی را بر روی آن انجام میدهد :

```
{
  result = argument0.arrObj[1];
  argument0.arrObj[1] = argument0.arrObj[argument0.numObj];
  argument0.arrObj[argument0.numObj] = 0;
  argument0.numObj -= 1;
  HeapifyDown(argument0, 1);
  return result;
}
```

قسمت بعدی کد اصلی بسادگی هشت جهت اصلی برای حرکت توپ را مشخص میکنند :

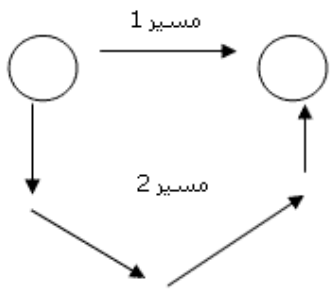
```
for (ii = 1; ii <= 8; ii += 1) {
  if (!place_meeting(arrPossDirectionsX[ii], arrPossDirectionsY[ii],
        objWall)) {
    newNode = instance_create(arrPossDirectionsX[ii],
                              arrPossDirectionsY[ii], dataNode);
    newNode.move = ii;
    newg = n.g + 1;
```

این حلقه For به ما اجازه میدهد تا در هر هشت جهت ممکن حرکت کنیم. (به سمت هشت گره ایجاد شده در اطراف موقعیت فعلی) در عین حال دستور Place_Meeting() باعث میشود تا از برخورد با دیوار ها جلوگیری به عمل آید. البته هنگام طراحی بازی تمامی این ساختار ها میتوانند دستخوش تغییر شوند. به عنوان مثال شما اجازه میدهید تا یک روح از میان دیوار ها عبور کند یا یک حیوان بتواند هم از میان خشکی و هم از میان آب عبور کند. اینها مواردی هستند که میبایست خودتان به ساختار الگوریتم اصلی اضافه کنید.

بنابراین در پایان حلقه چنانچه عملیات موفقیت آمیز بود ، یک گره جدید ایجاد میکنیم. سپس جهت حرکت را ذخیره میکنیم. بنابراین هر بار که یک خانه حرکت میکنیم یکی به متغیر مربوطه اضافه خواهیم کرد...

```
if (ListContains(dataList, newNode)) {
  if (ListFind(dataList, newNode).g <= newg)
     with (newNode) instance_destroy();
     else {
       newNode.parent = n;
       newNode.g = newg;
       newNode.h = Heuristic(newNode);
       newNode.f = newNode.g + newNode.h;
       if (ListContains(dataList, newNode))
         ListRemove(dataList, newNode);
       if (!ListContains(dataHeap, newNode))
         HeapAdd(dataHeap, newNode);
     }
```

توسط این قطعه کد بررسی میکنیم که آیا گره مورد نظر در لیست بسته قرار دارد یا خیر ... چرا ؟ به دیاگرام زیر دقت کنید :

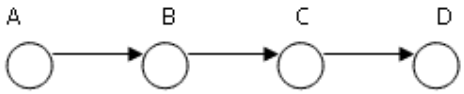


گره فرضی موجود در سمت چپ دو مسیر مختلف را ایجاد کرده است. یکی از این مسیر ها بطور مستقیم به گره هدف میرسد ، در حالی که مسیر دوم یک حلقه را ایجاد میکند و سپس به هدف میرسد. اکنون به بررسی اتفاقات رخ داده در طی دو مسیر میپردازیم. مشخص است که مقدار H در مسیر دوم بدون تغییر باقی مانده است. چرا که مقدارش به مختصات X و Y بستگی دارد که در اینجا در هر دو مسیر ثابت است. اما مقدار G آن (تعداد گامها برای رسیدن به مقصد) از مقدار G مسیر اول بسیار بیشتر است. به خاطر همین دلیل ما میتوانیم از مسیر هایی که

مقدار G بزرگی دارند براحتی چشم پوشی کنیم. چرا که میدانیم که آنها در واقع یک مسیر کمانی را طی کرده اند و مسلما مسیرهایی وجود دارند که مستقیم و یا دست کم مستقیم تر از این مسیر ها هستند. در عین حال این نکته در مورد گره هایی که اصلا ایجاد نمیشوند نیز صادق است. زیرا انها به این دلیل ایجاد نشده اند که از نظر جستجو گر گره های بهتری برای انتخاب وجود داشت. بنابراین آشکار نشدن این گره های فراموش شده هیچ مشکلی را ایجاد نخواهد کرد.

```
newNode.parent = n;
newNode.g = newg;
newNode.h = Heuristic(newNode);
newNode.f = newNode.g + newNode.h;
if (ListContains(dataList, newNode)) ListRemove(dataList, newNode);
if (!ListContains(dataHeap, newNode)) HeapAdd(dataHeap, newNode);
```

برای ایجاد یک گره جدید ، چند چیز مورد نیاز است. ما مقادیر G و H را فراهم کرده و سپس مقدار F را محاسبه میکنیم. در عین حال ما گره قبلی را به عنوان والد این گره جدید در نظر میگیریم. با این وجود خواهیم داشت :



همان گونه که مشاهد میشود c والد d و b والد c و a والد b است. در واقع با اینکار ما میتوانیم متوجه شویم که هر گره از کجا آمده است و در صورت نیاز منبع ایجاد یک گره دلخواه را پیدا کنیم.

چند خط کد باقی مانده در بدنه اصلی الگوریتم برای کامل کردن و برگشت دادن مقادیر مورد نیاز به کار میروند. بنابراین چنانچه ما تا انتهای کد پیش رفتیم و دست کم یک لیست بسته پیدا نکردیم ، به این معنی است که هیچ مسیری برای رسیدن از نقطه مبدا به نقطه مقصد موجود نبود و با برگشت دادن یک مقدار (در اینجا -1) این اتفاق را اعلام میکنیم.

حرکت کردن در مسیر مشخص شده :

بسیار خوب... تا به اینجا ما مسیر را مشخص کردیم... اما بعد چه ؟ در این قسمت به حرکت دادن شیء در مسیر مشخص شده میپردازیم :

```
  answerDirection = answer.move;
  answer = answer.parent;
  while (answer.parent != 0) {
    trailMarker = instance_create(answer.x, answer.y, objTrail);
    objMove.numNodes += 1; objMove.arrNodes[objMove.numNodes] =
answer;
    trailMarker.image_single = answerDirection-1;
    answerDirection = answer.move;
    answer = answer.parent;
  }
```

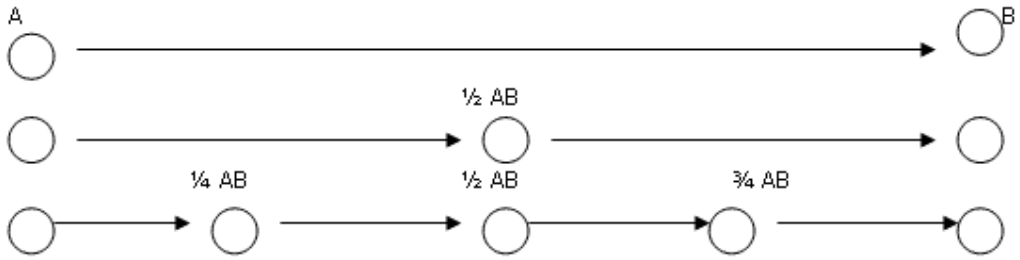
کاری که با این کد انجام میدهیم اینست که مرتبا والد ها را تا رسیدن به والد اصلی و اولیه (والد 0) دنبال میکنیم. در هر گام ما یک نشان گر کوچک اضافه میکنیم و گره مورد نظر را به آرایه ای که توپ میبایست توسط آن حرکت کند ، اضافه میکنیم. اما توجه داشته باشید چون عمل یافتن والد ما بصورت بر عکس است ، بنابراین عملگر آرایه نیز بصورت عکس شمار میشود و شما میبایست آنرا بر عکس کنید تا حرکت در جهت صحیح خود اتفاق بیافتد. در عین حال تابعی بنام Answer Detection افزوده شده است تا جهت گیری صحیح تضمین شود. هر یک از جهات

در مشخصات گره قبلی ذخیره شده اند. بنابراین برای اینکه از عملکرد صحیح الگوریتمتان اطمینان حاصل کنید این نکته مهم را به خاطر بسپارید.

نکات :

خسته نباشید. تا بحال می بایست درک ساده ای از این که الگوریتم *A چیست و چگونه کار میکند بدست آورده باشید. در ضمن بنظر میرسد که کد ها آنقدر ها هم که در ابتدا بنظر می آمدند مشکل و پیچیده نبودند... نظر شما چیست ؟ با این وجود تمامی نکاتی که در این مقاله به آن اشاره شده تئوری بودند و برای استفاده از آنها در یک بازی واقعی میبایست چند نکته اضافی را نیز در خاطر داشته باشید. اول از همه اینکه برای شخصی کردن این الگوریتم ، Heuristic و heap قسمتهایی هستند که در واقع باید تغییرات شخصی خود را در آنها اعمال کنید. در ضمن شما میتوانید ساختار های داده دیگری را برای ذخیره اطلاعات مربوط به نقاط و گره ها استفاده کنید. اگر چه این کار هنگامی که با نرم افزار GM مشغول ساخت این الگوریتم هستید کار آسانی نیست. در ضمن توجه داشته باشید که الگوریتمی که در مقاله بدان اشاره شد در مواقعی نادان عمل کرده و زیگ زاگ میرود و دلیل آن هم استفاده از یک عدد تخمینی بجای محاسبه دقیق وتر در هنگام حرکت قطری میباشد. برای رفع این مشکل میبایست از دستورات مربوط به ریشه یابی نظیر Sqrt() استفاده نمایید.

در اکثر موارد الگوریتم *A بی نقص و سریع عمل میکند. البته این بسیار خوب و مطلوب است. شما میتوانید شیء مورد نیاز خود را توسط این الگوریتم بخوبی به این سو و آن سو حرکت دهید. تنها مشکلی که در اینجا وجود دارد اینست که اگر تعداد واحد های حرکت کننده شما زیاد باشند ، دستورات محاسباتی میبایست برای همه اشیاء تکرار شوند و این باعث افت کیفیت بازی از حیث سرعت اجرا میگردد. برای رفع این مشکل میبایست الگوریتم را به تکه های کوچکتری برای بررسی تقسیم کنید. به مثال توجه کنید :



نکته مهمی که در این میان وجود دارد اینست که شما میتوانید یکباره گامهای بسیار بزرگی را برای مسیر یابی بردارید. (که البته باعث ایجاد یک مسیر خشن و پر پیچ و خم خواهد شد اما سریع به مقصد خواهد رسید) یا اینکه مسیر ها را به قسمتهای کوچکتری تقسیم کنید و بررسی ها را برای این مسیر های کوچک انجام دهید. در این صورت برنامه نیازی ندارد که مسیر کلی را برای همه اشیاء از ابتدا تا انتها محاسبه کند و فقط مسیر را تا نقطه مورد نیاز محاسبه میکند و بعد از اتمام این فرآیند به محاسبه مسیر های باقی مانده میپردازد. این کار باعث میشود که مسیر هموارتری برای حرکت کردن بدست بیاورید ، در عین حال که استفاده مضاعف از ذخایر سیستم تا حد زیادی بر طرف خواهد شد و هیچ مشکلی بر اثر روند مسیر یابی بر سرعت اجرای بازی تحمیل نخواهد شد.

در انتها این نکته را در ذهن داشته باشید که این الگوریتم تنها الگوریتم مسیر یابی موجود نیست و الگوریتم های زیادی وجود دارند که بر حسب نیاز شما ممکن است هم ساده تر باشند و هم بهتر پاسخ بدهند. بنابراین باز هم اختیار با شماست تا موردی که نیاز دارید را انتخاب کنید. البته در مورد این موضوع خاص نهایت دقت و وسواس را بخرج دهید تا بهترین گزینه را انتخاب کنید. اشتباه در انتخاب گزینه مناسب به ساختار بازی شما لطمه جبران ناپذیری خواهد زد...

" دو راه اساسی در طراحی یک نرم افزار وجود دارد. یکی اینکه آنرا به قدری ساده بسازید که هیچ اشکالی در آن یافت نشود... و دیگری آنکه آنرا بقدری پیچیده بسازید که هیچ اشکال آشکاری در آن یافت نشود... و البته راه اول بسیار مشکل تر از راه دوم است..."

سایت طراحان ایرانی با هدف آموزش ساخت بازیهای کامپیوتری به زبان فارسی طراحی شده است و تا کنون مقالات متعددی در زمینه های مختلف برنامه نویسی و ساخت بازی در آن قرار گرفته است. مدیریت سایت از تمامی عزیزان علاقمند به بازی های کامپیوتری ، برنامه نویسان ، طراحان و سایر کسانی که به نحوی با بازی ها در ارتباطند ، دعوت به همکاری به عمل می آورد تا بدینوسیله یک پایگاه علمی و موثق در زمینه صنعت ساخت بازیهای کامپیوتری در ایران ایجاد گردد.

در ضمن بسیاری از نرم افزار های ساخت بازی های کامپیوتری که امروزه در سطح وسیع مورد استفاده قرار میگیرند ، در سایت جمع آوری شده است و با مبلغ بسیار ناچیزی در اختیار علاقمندان به طراحی بازی های کامپیوتری قرار داده شده است. استفاده از این نرم افزار ها در آغاز کار و به منظور آشنا شدن با اصول اولیه در طراحی بازیها بسیار موثر و مفید بوده و شما میتوانید تا با چند جستجوی ساده در این زمینه ، به صحت موضوع پی ببرید. لیست زیر برخی از نرم افزار هایی هستند که توسط فروشگاه الکترونیکی سایت به مشتاقان عرضه میشوند :

- Game Maker Version 5.0 – 5.1 – 5.2 – 5.3 – 6.0 (Registered)
- The Game Factory (Home – Professional) (registered)
- Xtereme 3D 1.0
- King Space 3D
- Genesis 3D V1.6
- 3D Game Studio 5.12
- 3D State (Morfit 3D) (Registered)
- Blender 3D
- Q3d (Unregistered)
- Alice 3D
- True Vision 3D V6.2
- DirectX 9.0 Complete SDK (Software Development Kit Package)

و کامپایلر های

- Visual Basic V6.0
- Visual C++ V6.0

لینک فروشگاه الکترونیکی سایت طراحان ایرانی :

WWW.Persian-Designers.COM/index.php?pid=1
