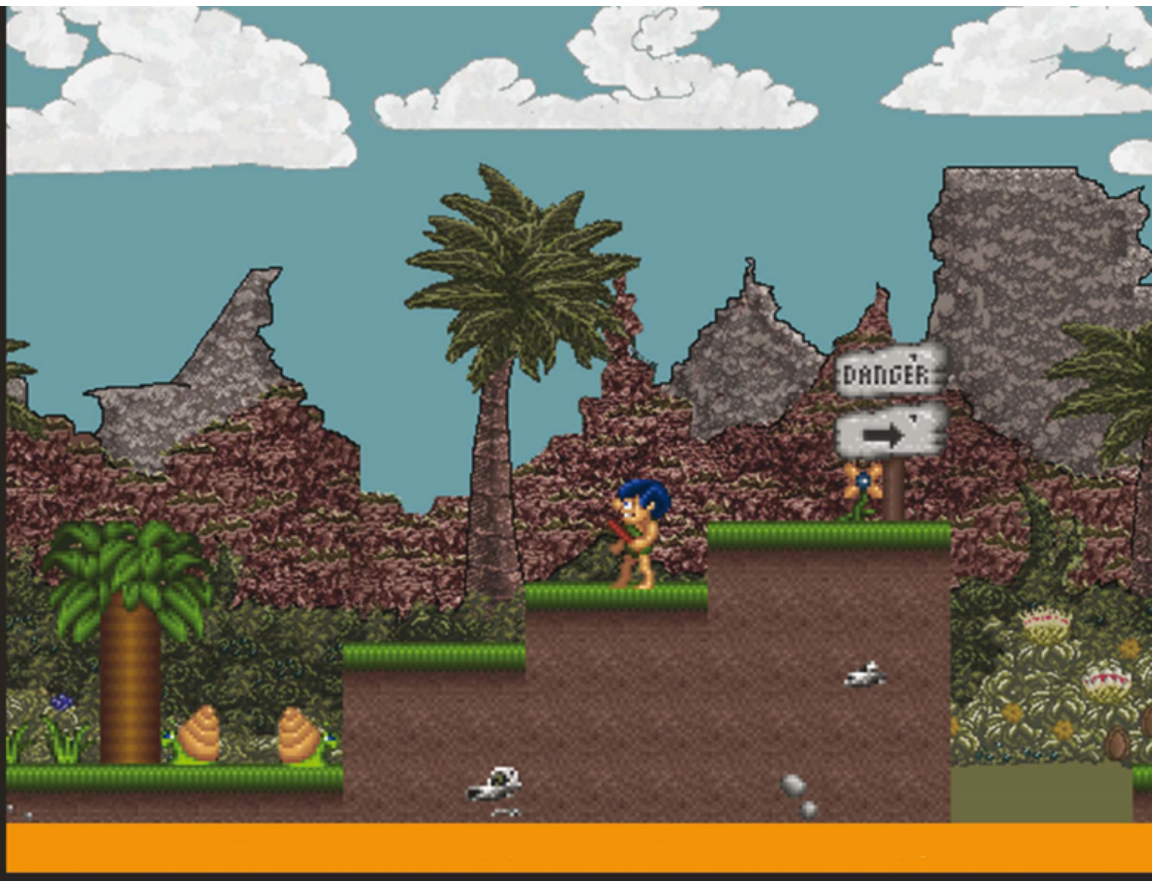


ساخت بازی با

SDL 2

کشف چگونگی استفاده از قدرت SDL 2.0
برای ایجاد بازی های بسیار جذاب در C++

۴

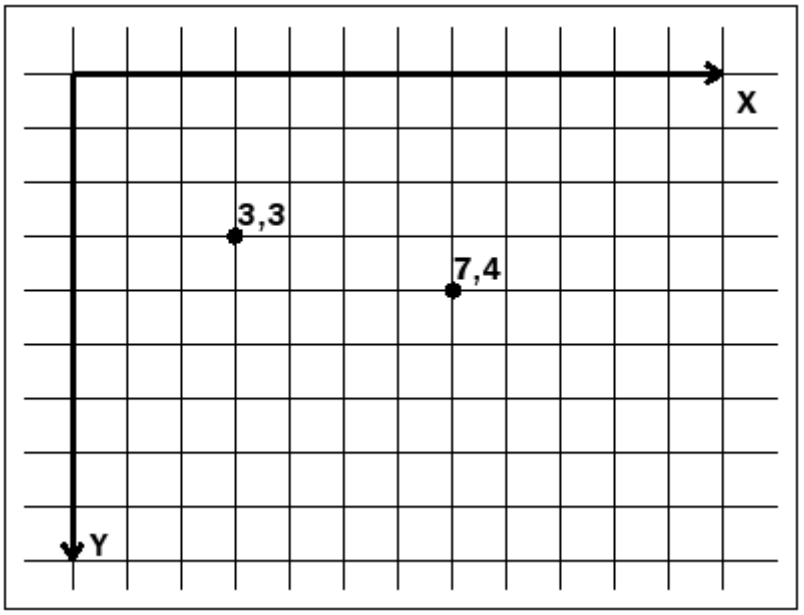
بررسی حرکت و مدیریت ورودی

ما پیش از این، ترسیم در صفحه نمایش و چگونگی مدیریت شی‌ها را پوشش دادیم اما هنوز چیزی نداشته ایم که به اطراف حرکت کند. گرفتن ورودی از کاربر و کنترل شی‌های بازی ما یکی از مهمترین مباحث در توسعه بازی است. این می‌تواند باعث ادراک و واکنش پذیری بازی شما شود و چیزی است که در حقیقت یک کاربر می‌تواند درک کند. در این فصل موارد زیر را پوشش خوهیم داد:

- سیستم مختصات کارتزین
- بردارهای دو بعدی
- ساختن متغیرها برای کنترل حرکت شی بازی
- برپا کردن یک سیستم حرکت ساده
- برپا کردن مدیریت ورودی از جوی‌استیک، صفحه‌کلید و ماوس
- ساختن یک نرخ فریم ثابت

تنظیم کردن شی‌های بازی برای حرکت

در فصل قبل، به شی‌های بازی خود مقدارهای x و y را دادیم که سپس می‌توانستیم به کد ترسیم خود ارسال کنیم. مقدارهای x و y که استفاده کردیم می‌توانند با استفاده از سیستم مختصات کارتزین(دکارتی) ارائه شوند.

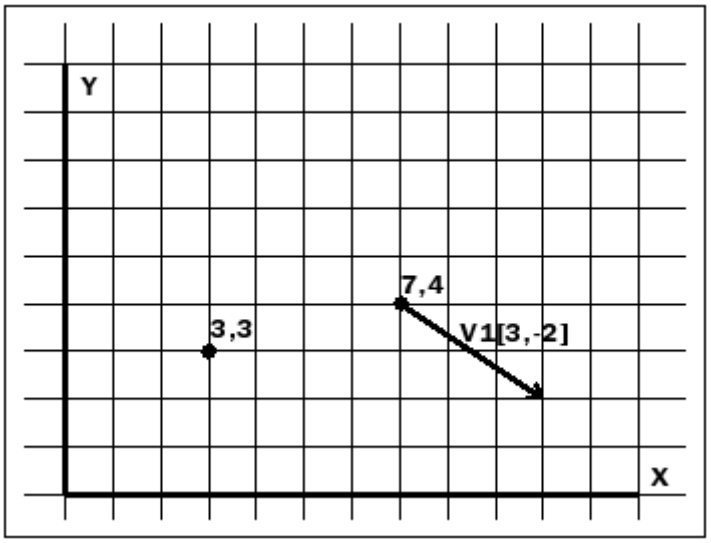


شکل بالا یک سیستم مختصات کارتزین را با دو مختصات نشان می‌دهد(روی محور Y معکوس شده است). ارائه آنها به صورت (x,y) به ما موقعیت 1 را به صورت (3,3) و موقعیت 2 را به صورت (7,4) می‌دهد.
این مقدارها می‌توانند برای ارائه دادن یک موقعیت در فضای دوبعدی استفاده شوند. این شکل را به صورت تصویر بزرگنمایی‌شده از گوشه چپ و بالا از پنجره بازی خود، با هر مربع شبکه که ارائه دهنده یک پیکسل از پنجره بازی خودمان است تصور کنید(در نظر بگیرید). با در نظر گرفتن این، می‌توانیم بفهمیم چگونه از این مقدارها برای ترسیم چیزها در صفحه نمایش در موقعیت درست استفاده کنیم. حالا ما به یک روش برای به روز رسانی مقدارهای این موقعیت نیاز داریم تا بتوانیم شی‌ها را به اطراف حرکت دهیم. برای انجام این کار ما بردارهای دو بعدی را بررسی خواهیم کرد.

یک بردار(vector) چیست؟

یک **بردار** می‌تواند به صورت یک موجودیت(entity) با یک جهت(direction) و یک بزرگی(magnitude) توصیف شود. می‌توانیم از آنها برای ارائه جنبه های شی‌های بازی خود استفاده کنیم، برای مثال، سرعت و شتاب، که می‌تواند برای ایجاد حرکت استفاده شود. با در نظر گرفتن سرعت

به عنوان مثال، برای ارائه دادن کامل سرعت شی‌های ما، به جهتی نیاز داریم که آنها در آن حرکت می‌کنند و همچنین مقداری(amount) (یا بزرگی) که از طریق آن در آن مسیر حرکت می‌کنند.



بیایید چند چیز در مورد اینکه چگونه از بردارها استفاده خواهیم کرد تعریف کنیم:

- ما یک بردار را به صورت v(x,y) نشان می‌دهیم
- ما می‌توانیم طول یک بردار را با استفاده از معادله زیر بدست آوریم:

$$lengthofv(x,y) = \sqrt{(x^2 + y^2)}$$

شکل قبل بردار v1(3,-2) را نشان می‌دهد که طول $\sqrt{(3^2+((-2)^2)}$ را خواهد داشت. ما می‌توانیم از اجزای x و y یک بردار برای نشان دادن موقعیت شی‌های خود در فضای دو بعدی استفاده کنیم. سپس می‌توانیم از چند عمل برداری متداول برای حرکت شی‌ها استفاده کنیم. قبل از اینکه سراغ این عملیات برویم بیایید یک کلاس بردار با نام Vector2D در پروژه بسازیم. سپس می‌توانیم هر عمل را که نیاز داریم بررسی کنیم و آن‌ها را به کلاس اضافه کنیم.

```
#include<math.h>
class Vector2D
{
public:
    Vector2D(float x, float y):
    m_x(x), m_y(y) {}
    float getX()
    {
        return m_x;
    }
    float getY()
    {
        return m_y;
    }
    void setX(float x)
    {
        m_x = x;
    }
    void setY(float y)
    {
        m_y = y;
    }
private:
    float m_x;
    float m_y;
};
```

شما می‌توانید ببینید که کلاس Vector2D در این جا بسیار ساده است. ما مقادیر x و y خود را داریم و راهی برای گرفتن(get) و تنظیم(set) آن‌ها داریم. ما حالا می‌دانیم که چگونه طول یک بردار را بدست آوریم، پس بیایید برای این هدف یک تابع ایجاد کنیم:

```
float length() { return sqrt(m_x * m_x + m_y * m_y); }
```

چند عمل متداول

اکنون از آنجایی که ما یک کلاس پایه داریم، می‌توانیم به تدریج برخی عملیات را اضافه کنیم.

جمع دو بردار

اولین عملی که ما آن را بررسی خواهیم کرد جمع دو بردار است. برای این کار ما به سادگی اجزای مجزای هر بردار را به هم اضافه می‌کنیم.

$$v_3 = v_1 + v_2 = (x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

بیایید از عملگرهای سربارگذاری‌شده استفاده کنیم تا جمع دو بردار با هم را برای ما آسان کند:

```
Vector2D operator+(const Vector2D& v2) const
{
    return Vector2D(m_x + v2.m_x, m_y + v2.m_y);
}

friend Vector2D& operator+=(Vector2D& v1, const Vector2D& v2)
{
    v1.m_x += v2.m_x;
    v1.m_y += v2.m_y;
    return v1;
}
```

با این توابع می‌توانیم با استفاده از عملگرهای جمع استاندارد دو بردار را به هم اضافه کنیم، به عنوان مثال:

```
Vector2D v1(10, 11);
Vector2D v2(35,25);
v1 += v2;
Vector2D v3 = v1 + v2;
```

ضرب در یک عدد نرده ای(scalar)

یک عمل دیگر ضرب کردن یک بردار در یک عدد اسکالر معمولی است.
برای این عمل هر جزء از بردار را در عدد اسکالر ضرب می‌کنیم:

$$v_1 * n = (x_1 * n, y_1 * n)$$

ما می‌توانیم برای ایجاد این توابع دوباره از عملگرهای سربارگذاری‌شده استفاده کنیم:

```
Vector2D operator*(float scalar)
{
   return Vector2D(m_x * scalar, m_y * scalar);
}

Vector2D& operator*=(float scalar)
{
   m_x *= scalar; m_y *= scalar;
   return *this;
}
```

تفریق دو بردار

تفریق بسیار شبیه به جمع است.

$$v_3 = v_1 - v_2 = (x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2)$$

بیایید چند تابع ایجاد کنیم تا این کار را برای ما انجام دهد:

```
Vector2D operator-(const Vector2D& v2) const
{
   return Vector2D(m_x - v2.m_x, m_y - v2.m_y);
}
```

```
friend Vector2D& operator-=(Vector2D& v1, const Vector2D& v2)
{
   v1.m_x -= v2.m_x;
   v1.m_y -= v2.m_y;
   return v1;
}
```

تقسیم بر یک عدد اسکالر

در حال حاظر من اطمینان دارم شما به الگوی آشکار شده پی برده اید و می‌توانید حدس بزنید تقسیم یک بردار بر یک اسکالر چگونه کار خواهد کرد، اما ما به هر حال این را پوشش خواهیم داد.

$$\frac{v_1}{n} = \left(\frac{x_1}{n}, \frac{y_1}{n}\right)$$

و توابع ما:

```
Vector2D operator/(float scalar)
{
   return Vector2D(m_x / scalar, m_y / scalar);
}

Vector2D& operator/=(float scalar)
{
   m_x /= scalar;
   m_y /= scalar;
   return *this;
}
```

نرمال سازی یک بردار

ما به عمل بسیار مهم دیگری نیاز داریم وآن قابلیت نرمال سازی یک بردار است. نرمال سازی یک بردار طول آن را هم پایه ای از 1 می‌کند. بردارهایی با طولی(بزرگی) از 1 به عنوان بردارهای یکه(واحد) شناخته می‌شود و تنها

برای نشان دادن یک جهت، از قبیل جهت مقابل یک شی مفید هستند. برای نرمال سازی یک بردار ما آن را در وارونش ضرب می‌کنیم.

$$l = length.\ v_{normalized} = v_1 * 1/l$$

ما می‌توانیم یک تابع عضو جدید برای نرمال سازی بردارهایمان ایجاد کنیم:

```
void normalize()
{
   float l = length();
   if ( l > 0) // we never want to attempt to divide by 0
   {
      (*this) *= 1 / l;
   }
}
```

حالا که چند تابع پایه داریم، بیایید شروع به استفاده از این توابع در کلاس SDLGameObject خود کنیم.

افزودن کلاس Vector2D

1. SDLGameObject.h را باز کنید و ما می‌توانیم پیاده سازی بردارها را شروع کنیم.

```
#include "Vector2D.h"
```

2. ما همچنین باید مقدارهای m_x و m_y را برداریم و آنها را با Vector2D جایگزین کنیم.

```
Vector2D m_position;
```

3. حالا ما می‌توانیم به فایل SDLGameObject.cpp برویم و سازنده را بروز رسانی کنیم.

```
SDLGameObject::SDLGameObject(const LoaderParams*
pParams) : GameObject(pParams), m_position(pParams->getX(),
pParams->getY())
{
    m_width = pParams->getWidth();
    m_height = pParams->getHeight();
    m_textureID = pParams->getTextureID();

    m_currentRow = 1;
    m_currentFrame = 1;
}
```

4. حالا ما بردار m_position را با استفاده از فهرست مقداردهی اولیه عضو ساختیم و ما همچنین باید از بردار m_position در تابع ترسیم خود استفاده کنیم.

```
void SDLGameObject::draw()
{
    TextureManager::Instance()->drawFrame(m_textureID,
(int)m_position.getX(), (int)m_position.getY(), m_width, m_height,
m_currentRow, m_currentFrame, TheGame::Instance()->getRenderer());
}
```

5. آخرین چیز قبل از اینکه آزمایش کنیم این است که از بردار خودمان در تابع Enemy::update استفاده کنیم.

```
void Enemy::update()
{
    m_position.setX(m_position.getX() + 1);
    m_position.setY(m_position.getY() + 1);
}
```

این تابع به زودی از جمع برداری استفاده خواهد کرد، اما در حال حاضر ما فقط ۱ را به موقعیت فعلی اضافه می‌کنیم تا همان رفتاری را داشته باشیم که

قبلا داشتیم. ما اکنون می‌توانیم این بازی را اجرا کنیم و خواهیم دید که یک سیستم برداری بسیار اساسی را پیاده سازی کرده‌ایم. به جلو بروید و با توابع Vector2D بازی کنید.

اضافه کردن سرعت

ما قبلا باید مقادیر x و y را به طور جداگانه تنظیم می‌کردیم، اما حالا که موقعیت ما یک بردار است، ما قابلیت اضافه کردن یک بردار جدید به آن برای به روز رسانی حرکت مان را داریم. ما این بردار را بردار سرعت می‌نامیم و می‌توانیم آن را به عنوان مقداری که می‌خواهیم شی مان در یک جهت خاص حرکت کند در نظر بگیریم:

1. بردار سرعت را می‌توان به صورت زیر نشان داد:

$$v_{position}+v_{velocity}= (x_{position}+x_{velocity} , y_{position}+ y_{velocity})$$

2. ما می‌توانیم این را به تابع update، SDLGameObject اضافه کنیم، چرا که این روشی است که ما همه اشیا مشتق‌شده را به روز می‌کنیم. بنابراین ابتدا اجازه دهید متغیر عضو سرعت را ایجاد کنیم.

```
Vector2D m_velocity;
```

3. ما آن را در لیست مقداردهی اولیه عضو به صورت ۰، ۰ می‌سازیم.

```
SDLGameObject::SDLGameObject(const LoaderParams* pParams) :
GameObject(pParams), m_position(pParams->getX(), pParams-
>getY()), m_velocity(0,0)
```

4. و حالا به تابع SDLGameObject::update می‌رسیم.

```
void SDLGameObject::update()
{
   m_position += m_velocity;
}
```

5. ما می‌توانیم این را در یکی از کلاس‌های مشتق‌شده تست کنیم. به Player.cpp بروید و موارد زیر را اضافه کنید:

```
void Player::update()
{
   m_currentFrame = int(((SDL_GetTicks() / 100) % 6));
   m_velocity.setX(1);
   SDLGameObject::update();
}
```

ما مقدار x، m_velocity را روی ۱ تنظیم کردیم. این یعنی اینکه هر بار که تابع update فراخوانی می‌شود ۱ را به مقدار x، m_position اضافه می‌کنیم. اکنون ما می‌توانیم این را اجرا کنیم تا ببینیم که شی ما با استفاده از این بردار جدید سرعت، حرکت می‌کند.

اضافه کردن شتاب

همه اشیا ما با سرعت ثابتی حرکت نمی‌کنند. برخی از بازی‌ها نیاز دارند که ما به تدریج سرعت جسم خود را با استفاده از شتاب افزایش دهیم. یک ماشین یا سفینه فضایی نمونه‌های خوبی هستند. هیچ‌کس انتظار نخواهد داشت که این اشیا فورا به حداکثر سرعت خود برسند. ما نیاز به یک بردار جدید برای شتاب داریم، پس بیایید این را به فایل SDLGameObject.h اضافه کنیم.

```
Vector2D m_acceleration;
```

سپس می توانیم این را به تابع update خود اضافه کنیم.

```
void SDLGameObject::update()
{
   m_velocity += m_acceleration;
   m_position += m_velocity;
 }
```

حالا تابع Player::update خود را تغییر دهید تا به جای سرعت، شتاب را تنظیم کنید.

```
void Player::update()
{
    m_currentFrame = int(((SDL_GetTicks() / 100) % 6));
    m_acceleration.setX(1);
    SDLGameObject::update();
}
```

بعد از اجرای بازی متوجه خواهید شد که شی به تدریج سرعت می‌گیرد.

ساختن فریم های ثابت در ثانیه

قبلا در این کتاب ما یک تابع SDL_Delay را برای آهسته کردن همه چیز و اطمینان از اینکه اشیا ما بیش از حد سریع حرکت نمی‌کنند، قرار دادیم. ما در حال حاضر با وادار کردن بازی خود به اینکه در یک نرخ فریم ثابت اجرا شود، آن را گسترش خواهیم داد. فریم های ثابت در ثانیه(fixed FPS) لزوما گزینه خوبی نیست، به خصوص زمانی که بازی شما شامل فیزیک های پیشرفته‌تر بیشتر می‌شود. ارزشش را دارد این را در ذهن داشته باشید برای وقتی که از این کتاب گذر کردید و شروع به توسعه بازی‌های خود کردید. با این حال FPS ثابت برای بازی‌های ۲ بعدی کوچک خوب خواهد بود، که ما در این کتاب برای رسیدن به آن تلاش خواهیم کرد.
با این گفته ها، بیایید به سراغ کد برویم:

1. main.cpp را باز کنید و ما چند متغیر ثابت را ایجاد خواهیم کرد.

```
const int FPS = 60;
const int DELAY_TIME = 1000.0f / FPS;
int main()
{
```

2. در اینجا ما مشخص می‌کنیم که در چه تعداد فریم در ثانیه می‌خواهیم بازی مان را اجرا کنیم. نرخ فریم ۶۰ فریم در ثانیه

نقطه شروع خوبی است چون اساسا با نرخ نوسازی(refresh) بیشتر مانیتورها و تلویزیون‌های مدرن موجود هماهنگ است. سپس می‌توانیم این را به تعداد میلی ثانیه در یک ثانیه تقسیم کنیم، و به ما مقدار زمانی را می‌دهد که نیاز داریم تا بازی را بین حلقه‌ها به تاخیر بیندازیم تا نرخ فریم ثابت خود را حفظ کنیم. ما به دو متغیر دیگر در بالای تابع main خود نیاز داریم؛ اینها در محاسبات ما مورد استفاده قرار خواهند گرفت.

```
int main()
{
   Uint32 frameStart, frameTime;
```

3. ما اکنون می‌توانیم نرخ فریم ثابت خود را در حلقه اصلی خود پیاده سازی کنیم.

```
while(TheGame::Instance()->running())
{
   frameStart = SDL_GetTicks();

   TheGame::Instance()->handleEvents();
   TheGame::Instance()->update();
   TheGame::Instance()->render();

   frameTime = SDL_GetTicks() - frameStart;

   if(frameTime< DELAY_TIME)
   {
      SDL_Delay((int)(DELAY_TIME - frameTime));
   }
}
```

ابتدا در ابتدای حلقه زمان را می‌گیریم و آن را در frameStart ذخیره می‌کنیم. برای این کار از SDL_GetTicks استفاده می‌کنیم که مقدار میلی ثانیه ها را از زمانی که SDL_init را فراخوانی کرده ایم باز

می‌گرداند. سپس حلقه بازی خود را اجرا می‌کنیم و با کم کردن زمانی که فریم ما شروع شده از زمان جاری، ذخیره می‌کنیم که چقدر طول می‌کشد تا (حلقه) اجرا شود. اگر کم‌تر از زمانی است که ما می خواهیم یک فریم طول بکشد، ما SDL_Delay را فرامی‌خوانیم و حلقه مان را برای مقدار زمانی که می‌خواهیم طول بکشد با کم کردن این که حلقه تا به حال چقدر طول کشیده است منتظر نگه می‌داریم تا کامل شود.

مدیریت ورودی

ما در حال حاضر اشیا متحرک خود را براساس سرعت و شتاب داریم، بنابراین سپس باید روش هایی برای کنترل این حرکت از طریق ورودی کاربر معرفی کنیم. SDL تعدادی از انواع مختلفی از رابط کاربری دستگاه ها از جمله جوی‌استیک، گیم پد، ماوس و صفحه‌کلید را پشتیبانی می‌کند، که همه آن‌ها در این بخش پوشش داده خواهند شد، به همراه این که چگونه آن‌ها را به پیاده‌سازی چارچوب خود اضافه کنیم.

ساختن کلاس مدیریت کننده ورودی خودمان

ما یک کلاس ایجاد خواهیم کرد که تمام ورودی‌های دستگاه را مدیریت کند، چه از کنترل‌کننده‌ها(controllers)، صفحه‌کلید، یا ماوس باشد. بیایید با یک کلاس پایه شروع کنیم و از اینجا پی ریزی میکنیم. ابتدا به یک فایل سرآیند، InputHandler.h نیاز داریم.

```
#include "SDL.h"
class InputHandler
{
public:
   static InputHandler* Instance()
   {
      if(s_pInstance == 0)
      {
         s_pInstance = new InputHandler();
```

```
        }
        return s_pInstance;
    }

    void update();
    void clean();

private:

    InputHandler();
    ~InputHandler() {}

    static InputHandler* s_pInstance;
};
typedef InputHandler TheInputHandler;
```

این InputHandler یگانه ما است. تا کنون ما یک تابع update داریم که رویدادها را دریافت خواهد کرد و InputHandler ما را براساس آن به روز رسانی خواهد کرد، و یک تابع clean هر دستگاهی را که ما راه‌اندازی کرده‌ایم را پاک می‌کند. همانطور که ما اضافه کردن پشتیبانی از دستگاه را شروع می‌کنیم، این کار را با جزییات خیلی بیشتر توضیح خواهیم داد.

مدیریت ورودی joystick/gamepad

در بیرون ده ها joystick و gamepad که اغلب با مقادیر متفاوتی از دکمه‌ها و دسته‌های آنالوگ هستند در میان چیزهای دیگر وجود دارد. توسعه دهندگان بازی‌های PC وقتی تلاش می‌کنند که از همه این gamepad های متفاوت پشتیبانی کنند باید کارهای زیادی انجام دهند. SDL برای joystick ها و gamepad ها پشتیبانی خوبی دارد، بنابراین ما باید قادر باشیم سیستمی را ارائه کنیم که برای گسترش پشتیبانی gamepad های مختلف دشوار نباشد.

رویدادهای جوی‌استیک SDL

چند ساختار مختلف برای مدیریت joystick در SDL وجود دارد. جدول زیر هر یک از آن‌ها و اهداف آن‌ها را فهرست می‌کند.

رویداد جوی‌استیک SDL	هدف
SDL_JoyAxisEvent	اطلاعات حرکت محور
SDL_JoyButtonEvent	اطلاعات فشرده و آزاد شدن دکمه
SDL_JoyBallEvent	اطلاعات حرکت رویداد توپک
SDL_JoyHatEvent	تغییرات موقعیت hat جوی‌استیک

رویدادهایی که بیش از همه به آن توجه داریم رویدادهای حرکت محور و رویدادهای فشردن دکمه هستند. هر یک از این رویدادها یک نوع شمارشی(enumerated) دارد که ما می‌توانیم در حلقه رویدادهای خود کنترل کنیم تا اطمینان حاصل شود که ما تنها به رویدادهایی رسیدگی می‌کنیم که می‌خواهیم مدیریت کنیم. جدول زیر مقدار نوع برای هر یک از رویدادهای بالا را نشان می‌دهد.

رویداد جوی‌استیک SDL	مقدار نوع
SDL_JOYAXISMOTION	SDL_JoyAxisEvent
SDL_JOYBUTTONDOWN یا SDL_JOYBUTTONUP	SDL_JoyButtonEvent
SDL_JOYBALLMOTION	SDL_JoyBallEvent
SDL_JOYHATMOTION	SDL_JoyHatEvent

استفاده از ویژگی **Joystick Control Panel** در ویندوز یا **JoystickShow** روی OSX برای پی بردن به اینکه کدام اعداد دکمه را شما نیاز خواهید داشت تا در SDL برای یک دکمه خاص استفاده کنید ایده خوبی است. این اپلیکیشن ها برای پی بردن به چیزهایی در مورد جوی‌استیک/گیم پد با ارزش هستند بنابراین شما می‌توانید آنها را به درستی پشتیبانی کنید.

کدی که ما مطرح خواهیم کرد فرض می‌گیریم از یک کنترل‌کننده مایکروسافت ایکس باکس 360 (که می‌تواند روی PC یا OSX استفاده شود) استفاده می‌کنیم با نظر به اینکه این یک کنترل‌کننده بسیار محبوب برای PC گیمینگ است. کنترل‌کننده های دیگر از قبیل، کنترل‌کننده PS3 احتمالا می‌تواند مقدارهای متفاوت برای دکمه ها و محورها داشته باشد. کنترل‌کننده ایکس باکس 360 شامل موارد زیر است:

- دو محور آنالوگ
- محورهای آنالوگ فشاری به شکل دکمه ها
- چهار دکمه جلو: A, B, X, Y

- چهار تریگر: دو دیجیتال و دو آنالوگ
- یک پد دیجیتال جهت

مقداردهی جوی‌استیک ها

1. برای استفاده از گیم پد ها و جوی‌استیک ها در SDL ما ابتدا باید آنها را مقداردهی کنیم. ما می‌خواهیم یک تابع عمومی(public) جدید را به کلاس InputHandler اضافه می‌کنیم.

```
void initialiseJoysticks();
bool joysticksInitialised() {
return m_bJoysticksInitialised; }
```

2. ما همچنین چند متغیر عضو که نیاز داریم را اعلام می‌کنیم.

```
std::vector<SDL_Joystick*> m_joysticks;
bool m_bJoysticksInitialised;
```

3. SDL_Joystick* یک اشاره گر به جوی‌استیکی است که ما مقداردهی خواهیم کرد. ما در حقیقت نیازی به این اشاره گرها هنگام استفاده از جوی‌استیک ها نداریم، اما نیاز داریم تا آنها را پس از اتمام کار ببندیم(close)، پس این برای ما مفید است تا فهرستی از آنها را برای دسترسی بعدی نگه داریم. ما اکنون تابع initialiseJoysticks خود را تعریف خواهیم کرد و سپس آن را بررسی می‌کنیم.

```
void InputHandler::initialiseJoysticks()
{
   if(SDL_WasInit(SDL_INIT_JOYSTICK) == 0)
   {
   SDL_InitSubSystem(SDL_INIT_JOYSTICK);
   }
   if(SDL_NumJoysticks() > 0)
   {
      for(int i = 0; i < SDL_NumJoysticks(); i++)
      {
         SDL_Joystick* joy = SDL_JoystickOpen(i);
```

```
            if(SDL_JoystickGetAttached(joy) == 1)
            {
                m_joysticks.push_back(joy);
            }
            else
            {
                std::cout << SDL_GetError();
            }
        }
        SDL_JoystickEventState(SDL_ENABLE);
        m_bJoysticksInitialised = true;
        std::cout << "Initialised "<< m_joysticks.size() << " joystick(s)";
    }
    else
    {
        m_bJoysticksInitialised = false;
    }
}
```

4. بیایید این را خط به خط بررسی کنیم. ابتدا با استفاده از SDL_WasInit بررسی می‌کنیم آیا زیرسیستم جوی‌استیک مقداردهی شده است. اگر مقداردهی نشده است سپس با استفاده از SDL_InitSubSystem آن را مقداردهی می‌کنیم.

```
if(SDL_WasInit(SDL_INIT_JOYSTICK) == 0)
{
        SDL_InitSubSystem(SDL_INIT_JOYSTICK);
}
```

5. سپس باز کردن هر جوی‌استیک در دسترس است. پیش از اینکه سعی کنیم شی‌ها را باز کنیم، از SDL_NumJoysticks برای اطمینان از وجود تعدادی جوی‌استیک در دسترس استفاده می‌کنیم. سپس می‌توانیم از میان تعداد جوی‌استیک ها گذر کنیم، به نوبت آنها را با

SDL_JoystickOpen باز کنیم. آنها سپس می‌توانند برای بعدها بسته شدن در آرایه ما قرار داده شوند.

```
if(SDL_NumJoysticks() > 0)
{
   for(int i = 0; i < SDL_NumJoysticks(); i++)
   {
      SDL_Joystick* joy = SDL_JoystickOpen(i);
      if(SDL_JoystickGetAttached(joy) == 1)
      {
         m_joysticks.push_back(joy);
      }
      else
      {
          std::cout << SDL_GetError();
      }
    }
}
```

6. در نهایت، ما به SDL می‌گوییم که با فعال‌سازی SDL_JoystickEventState ، شروع به گوش دادن(توجه) به رویدادهای جوی‌استیک کند. ما همچنین متغیر عضو m_bJoysticksEnabled خود را بر حسب اینکه مقداردهی ما چگونه پیش رفت تنظیم می‌کنیم.

```
SDL_JoystickEventState(SDL_ENABLE);
m_bJoysticksInitialised = true;

std::cout << "Initialised " << m_joysticks.size() << " joystick(s)";

}
else
{
   m_bJoysticksInitialised = false;
}
```

7. بنابراین، ما اکنون راهی برای راه‌اندازی جوی‌استیک های خود داریم. ما دو تابع دیگر، یعنی توابع update و clean را برای تعریف داریم. تابع clean از میان آرایه ی *SDL_Joystick گذر می‌کند و در هر تکرار SDL_JoystickClose را فراخوانی می‌کند.

```
void InputHandler::clean()
{
   if(m_bJoysticksInitialised)
   {
      for(unsigned int i = 0; i < SDL_NumJoysticks(); i++)
      {
         SDL_JoystickClose(m_joysticks[i]);
      }
   }
}
```

8. تابع update در هر فریم در حلقه اصلی بازی برای به روز رسانی وضعیت رویداد فراخوانی خواهد شد. در حال حاضر، به هر حال آن به سادگی به یک رویداد خروج گوش می‌دهد و تابع quit بازی را فراخوانی می‌کند (این تابع به سادگی SDL_Quit() را فراخوانی می‌کند).

```
void InputHandler::update()
{
   SDL_Event event;
   while(SDL_PollEvent(&event))
   {
      if(event.type == SDL_QUIT)
      {
         TheGame::Instance()->quit();
      }
   }
}
```

9. اکنون ما از این InputHandler در تابع کلاس Game خود استفاده خواهیم کرد. ابتدا initialiseJoysticks را در تابع Game::init فراخوانی می‌کنیم:

```
TheInputHandler::Instance()->initialiseJoysticks();
```

و ما آن را در تابع Game::handleEvents به روز خواهیم کرد: هر چیزی که از قبل داشتیم را پاک‌سازی می‌کنیم:

```
void Game::handleEvents()
{
   TheInputHandler::Instance()->update();
}
```

10. همچنین تابع clean می‌تواند به تابع Game::clean اضافه شود.

```
TheInputHandler::Instance()->clean();
```

11. حالا ما می‌توانیم یک پد یا جوی‌استیک را وصل کنیم و ساختن(build) را اجرا کنیم. اگر همه چیز طبق برنامه کار کند، ما باید خروجی زیر را با x نشان دهنده تعداد جوی‌استیک هایی که شما وصل کرده اید بگیریم:

```
Initialised x joystick(s)
```

12. به طور دلخواه ما می‌خواهیم به سادگی از یک یا تعداد بیشتری کنترل‌کننده بدون هیچ تغییری در کد خود استفاده کنیم. ما از قبل یک روش برای بارگذاری و باز کردن هر تعداد کنترل‌کننده که وصل شده است داریم، اما باید بدانیم هر رویداد مربوط به کدام کنترل‌کننده است: ما این را با تعدادی اطلاعات ذخیره شده در رویداد(event) انجام میدهیم. هر رویداد جوی‌استیک یک متغیر which خواهد داشت که در آن ذخیره می‌شود. با استفاده از این به ما اجازه خواهد داد که پی ببریم رویداد از کدام جوی‌استیک می‌آید.

```
if(event.type == SDL_JOYAXISMOTION) // check the type value
{
   int whichOne = event.jaxis.which; // get which controller
```

گوش دادن به حرکت محور و مدیریت آن

ما قرار نیست دسته های آنالوگ را به روش آنالوگ مدیریت کنیم.
در عوض آنها به صورت اطلاعات دیجیتال مدیریت خواهند شد، و به این شکل، آنها یا روشن(on) یا خاموش(off) هستند. کنترل‌کننده ما چهار محور حرکت، دوتا برای محور آنالوگ سمت چپ و دوتا برای محور آنالوگ سمت راست دارد.

ما مفروضات زیر را در مورد کنترل‌کننده خودمان مطرح می‌کنیم (شما می‌توانید از یک اپلیکیشن خارجی استفاده کنید تا به مقدارهای خاص برای کنترل‌کننده خودتان پی ببرید):

- حرکت چپ و راست روی stick اول محور 0 است
- حرکت بالا و پایین روی stick اول محور 1 است
- حرکت چپ و راست روی stick دوم محور 3 است
- حرکت بالا و پایین روی stick دوم محور 4 است

کنترل‌کننده ایکس باکس 360 از محورهای 2 و 5 برای تریگرهای آنالوگ استفاده می‌کند. برای مدیریت کردن چند کنترل‌کننده با چند محور ما یک بردار از جفت های *Vector2D ، یکی برای هر stick خواهیم ساخت.

```
std::vector<std::pair<Vector2D*, Vector2D*>> m_joystickValues;
```

ما از مقدارهای Vector2D برای تنظیم اینکه آیا یک stick به بالا، پایین، چپ، یا راست حرکت کرده است. حالا هنگامی که جوی‌استیک های خود را مقداردهی می‌کنیم ما باید یک جفت از *Vector2D در آرایه m_joystickValues بسازیم.

```
for(int i = 0; i < SDL_NumJoysticks(); i++)
{
```

```
    SDL_Joystick* joy = SDL_JoystickOpen(i);
    if(SDL_JoystickGetAttached(joy) == 1)
    {
        m_joysticks.push_back(joy);
        m_joystickValues.push_back(std::make_pair(new
        Vector2D(0,0),new Vector2D(0,0))); // add our pair
    }
    else
    {
        std::cout << SDL_GetError();
    }
}
```

ما به روشی نیاز داریم تا این مقدارهایی که نیاز داریم را از این آرایه از جفت ها بگیریم: ما در کلاس InputHandler دو تابع جدید اعلام می‌کنیم:

```
int xvalue(int joy, int stick);
int yvalue(int joy, int stick);
```

پارامتر joy شناسه(ID) جوی‌استیکی است که می‌خواهیم استفاده کنیم، و stick برای stick چپ 1 و برای stick راست 2 است. بیاید این توابع را تعریف کنیم:

```
int InputHandler::xvalue(int joy, int stick);
{
    if(m_joystickValues.size() > 0)
    {
        if(stick == 1)
        {
            return m_joystickValues[joy].first->getX();
        }
        else if(stick == 2)
        {
            return m_joystickValues[joy].second->getX();
        }
    }
```

```
    return 0;
}

int InputHandler::yvalue(int joy, int stick)
{
    if(m_joystickValues.size() > 0)
    {
        if(stick == 1)
        {
            return m_joystickValues[joy].first->getY();
        }
        else if(stick == 2)
        {
            return m_joystickValues[joy].second->getY();
        }
    }
    return 0;
}
```

پس ما مقدار x یا y را بر اساس پارامترهای ارسال شده به هر تابع می‌گیریم. مقدارهای first و second اولین یا دومین شی از جفت در آرایه هستند، joy نشان دهنده اندیس آرایه است. ما اکنون می‌توانیم در حلقه رویداد این مقدارها را به درستی تنظیم کنیم.

```
SDL_Event event;
while(SDL_PollEvent(&event))
{
    if(event.type == SDL_QUIT)
    {
        TheGame::Instance()->quit();
    }
    if(event.type == SDL_JOYAXISMOTION)
    {
        int whichOne = event.jaxis.which;
        // left stick move left or right
```

```
if(event.jaxis.axis == 0)
{
    if (event.jaxis.value > m_joystickDeadZone)
    {
        m_joystickValues[whichOne].first->setX(1);
    }
    else if(event.jaxis.value < -m_joystickDeadZone)
    {
        m_joystickValues[whichOne].first->setX(-1);
    }
    else
    {
        m_joystickValues[whichOne].first->setX(0);
    }
}
// left stick move up or down
if(event.jaxis.axis == 1)
{
    if (event.jaxis.value > m_joystickDeadZone)
    {
        m_joystickValues[whichOne].first->setY(1);
    }
   else if(event.jaxis.value < -m_joystickDeadZone)
   {
        m_joystickValues[whichOne].first->setY(-1);
    }
    else
    {
        m_joystickValues[whichOne].first->setY(0);
    }
}

// right stick move left or right
if(event.jaxis.axis == 3)
{
```

```
        if (event.jaxis.value > m_joystickDeadZone)
        {
            m_joystickValues[whichOne].second->setX(1);
        }
        else if(event.jaxis.value < -m_joystickDeadZone)
        {
            m_joystickValues[whichOne].second->setX(-1);
        }
        else
        {
            m_joystickValues[whichOne].second->setX(0);
        }
    }

    // right stick move up or down
    if(event.jaxis.axis == 4)
    {
        if (event.jaxis.value > m_joystickDeadZone)
        {
            m_joystickValues[whichOne].second->setY(1);
        }
        else if(event.jaxis.value < -m_joystickDeadZone)
        {
            m_joystickValues[whichOne].second->setY(-1);
        }
        else
        {
            m_joystickValues[whichOne].second->setY(0);
        }
    }
  }
}
```

این یک تابع بزرگ است، هر چند نسبتا ساده است. ما ابتدا برای یک رویداد SDL_JOYAXISMOITON بررسی انجام میدهیم و سپس با استفاده از مقدار which به اینکه رویداد از کدام کنترل‌کننده آمده است پی میبریم.

```
int whichOne = event.jaxis.which;
```

از اینجا می‌دانیم رویداد از کدام جوی‌استیک آمده است و می‌توانیم یک مقدار متناسب را در آرایه تنظیم کنیم. برای نمونه:

```
m_joystickValues[whichOne]
```

ابتدا محوری که از آن رویدادی آمده است را بررسی می‌کنیم:

```
if(event.jaxis.axis == 0) // …1,3,4
```

اگر محور(axis) صفر یا یک است، محور چپ است و اگر سه یا چهار است، محور راست است. ما از first یا second از جفت برای تنظیم محور چپ یا راست استفاده می‌کنیم. شما همچنین شاید متوجه متغیر m_joystickDeadZone شده باشید. ما از این برای حساب کردن میزان حساسیت(sensitivity) یک کنترل‌کننده استفاده می‌کنیم. ما می‌توانیم این را به صورت یک متغیر ثابت در فایل سرآیند InputHandler تنظیم کنیم:

```
const int m_joystickDeadZone = 10000;
```

مقدار 10000 ممکن است یک مقدار بزرگ به نظر رسد که برای یک محور استفاده شود، اما میزان حساسیت یک کنترل‌کننده می‌تواند بسیار بالا باشد و بنابراین به یک مقدار به این بزرگی نیاز دارد. این مقدار را متناسب با کنترل‌کننده خودتان تغییر دهید.

فقط برای یکپارچه شدن چیزی که اینجا انجام دادیم، بیایید به دقت به یک حالت نگاهی بیندازیم.

```
// left stick move left or right
{
   if (event.jaxis.value > m_joystickDeadZone)
```

```
  {
     m_joystickValues[whichOne].first->setX(1);
   }
   else if(event.jaxis.value < -m_joystickDeadZone)
   {
   m_joystickValues[whichOne].first->setX(-1);
   }
   else
   {
     m_joystickValues[whichOne].first->setX(0);
   }
}
```

اگر ما به دومین دستور if برسیم، میدانیم که با یک رویداد حرکت چپ یا راست روی stick چپ سر و کار داریم به دلیل اینکه axis(محور) 0 بوده است. ما از قبل تنظیم کرده ایم که رویداد از کدام کنترل‌کننده آمده است و whichOne را روی مقدار درست تنظیم می‌کنیم. ما همچنین می‌خواهیم first از جفت stick چپ باشد. ما از اولین(first) شی از آرایه استفاده می‌کنیم و مقدار x آن را تنظیم می‌کنیم چنانکه با یک رویداد حرکت x سروکار داریم. پس چرا ما مقدار را روی 1 یا 1- تنظیم کردیم؟ با شروع حرکت دادن شی بازیکن (Player) خود به این پاسخ خواهیم داد.

Player.h را باز کنید و ما می‌توانیم شروع به استفاده از InputHandler برای گرفتن رویدادها کنیم. ابتدا یک تابع خصوصی(private) جدید اعلام خواهیم کرد:

```
private:

void handleInput();
```

حالا در فایل Player.h خود، می‌توانیم این تابع را تعریف کنیم تا با InputHandler کار کند.

```
void Player::handleInput()
{
    if(TheInputHandler::Instance()->joysticksInitialised())
    {
        if(TheInputHandler::Instance()->xvalue(0, 1) > 0 ||
        TheInputHandler::Instance()->xvalue(0, 1) < 0)
        {
                m_velocity.setX(1 * TheInputHandler::Instance()->xvalue(0,
                                1));
        }

        if(TheInputHandler::Instance()->yvalue(0, 1) > 0 ||
        TheInputHandler::Instance()->yvalue(0, 1) < 0)
        {
                m_velocity.setY(1 * TheInputHandler::Instance()->yvalue(0,
                                1));
        }

        if(TheInputHandler::Instance()->xvalue(0, 2) > 0 ||
        TheInputHandler::Instance()->xvalue(0, 2) < 0)
        {
                m_velocity.setX(1 * TheInputHandler::Instance()->xvalue(0,
                2));
        }

        if(TheInputHandler::Instance()->yvalue(0, 2) > 0 ||
        TheInputHandler::Instance()->yvalue(0, 2) < 0)
        {
                m_velocity.setY(1 * TheInputHandler::Instance()->yvalue(0,
                                2));
        }
    }
}
```

سپس ما می‌توانیم این تابع را در تابع Player::update فرا بخوانیم.

```
void Player::update()
{
	m_velocity.setX(0);
	m_velocity.setY(0);
	handleInput(); // add our function
	m_currentFrame = int(((SDL_GetTicks() / 100) % 6));
	SDLGameObject::update();
}
```

الان همه چیز درست است، اما ابتدا بیایید بررسی کنیم ما چگونه حرکت خود را تنظیم می‌کنیم.

```
if(TheInputHandler::Instance()->xvalue(0, 1) > 0 ||
TheInputHandler::Instance()->xvalue(0, 1) < 0)
{
   m_velocity.setX(1 * TheInputHandler::Instance()->xvalue(0, 1));
}
```

اینجا، ما ابتدا بررسی می‌کنیم آیا xvalue از stick چپ بزرگتر از 0 است (که حرکت کرده است). اگر چنین است ما x سرعت Player را روی مقداری که می‌خواهیم در xvalue از stick چپ که می‌دانیم یا 1 یا 1- است ضرب شود تنظیم می‌کنیم. چنانکه شما می‌دانید، ضرب یک عدد مثبت در یک عدد منفی، یه یک عدد منفی منتج می‌شود، پس ضرب، سرعتی که می‌خواهیم در
منفی یک یعنی ما x سرعت خود را روی یک مقدار منفی تنظیم کرده ایم (به چپ حرکت می‌کند). ما همین را برای دسته دیگر و همچنین مقدارهای y انجام می‌دهیم. پروژه را Build کنید و با یک گیم پد شروع به حرکت دادن شی Player خود کنید. شما همچنین می‌توانید کنترل‌کننده دیگری را وصل کنید و کلاس Enemy را برای استفاده از آن به روز رسانی کنید.

تعامل با ورودی دکمه جوی‌استیک

گام بعدی پیاده‌سازی روشی برای مدیریت ورودی دکمه از کنترل‌کننده ها می‌باشد. این در واقع بسیار ساده‌تر از کنترل محورها است. ما باید وضعیت فعلی هر دکمه را بدانیم که می‌توانیم هر گاه یکی فشرده یا آزاد شد را بررسی کنیم. برای انجام این کار، ما یک آرایه از مقادیر بولی را اعلام خواهیم کرد، بنابراین هر کنترل‌کننده (اولین اندیس در آرایه) یک آرایه از مقادیر بولی، یکی برای هر دکمه روی کنترل‌کننده خواهد داشت.

```
std::vector<std::vector<bool>> m_buttonStates;
```

ما می‌توانیم وضعیت دکمه فعلی را با یک تابع که دکمه درست از جوی‌استیک درست را بررسی می‌کند، بدست آوریم.

```
bool getButtonState(int joy, int buttonNumber)
{
   return m_buttonStates[joy][buttonNumber];
}
```

اولین پارامتر اندیس آرایه (شناسه جوی‌استیک) است، و دومی اندیس دکمه‌ها است. سپس باید این آرایه را برای هر کنترل‌کننده و هر یک از دکمه‌های آن مقداردهی کنیم. ما این کار را در تابع initialiseJoysticks انجام خواهیم داد.

```
for(int i = 0; i < SDL_NumJoysticks(); i++)
{
   SDL_Joystick* joy = SDL_JoystickOpen(i);
   if(SDL_JoystickOpened(i))
   {
      m_joysticks.push_back(joy);
      m_joystickValues.push_back(std::make_pair(new
      Vector2D(0,0),new Vector2D(0,0)));

      std::vector<bool> tempButtons;

      for(int j = 0; j < SDL_JoystickNumButtons(joy); j++)
      {
```

```
            tempButtons.push_back(false);
        }

        m_buttonStates.push_back(tempButtons);
    }
}
```

از SDL_JoystickNumButtons برای بدست آوردن تعداد دکمه‌ها برای هر یک از جوی‌استیک ها استفاده می‌کنیم. سپس یک مقدار برای هر کدام از این دکمه‌ها در یک آرایه قرار می‌دهیم. برای شروع false را قرارمی‌دهیم، چون هیچ دکمه ای فشرده نمی‌شود. سپس این آرایه در آرایه m_buttonStates ما قرار می‌گیرد تا با تابع getButtonState استفاده شود. حال باید به رویدادهای دکمه گوش دهیم و مقدار را در آرایه متناسب با آن تنظیم کنیم.

```
if(event.type == SDL_JOYBUTTONDOWN)
{
    int whichOne = event.jaxis.which;
    m_buttonStates[whichOne][event.jbutton.button] = true;
}
if(event.type == SDL_JOYBUTTONUP)
{
    int whichOne = event.jaxis.which;
    m_buttonStates[whichOne][event.jbutton.button] = false;
}
```

هنگامی که یک دکمه فشار داده می‌شود (SDL_JOYBUTTONDOWN) باید بدانیم این روی کدام کنترل‌کننده فشرده شده است و از این به عنوان اندیسی در آرایه m_buttonStates استفاده می‌کنیم. سپس از شماره دکمه (event.jbutton.button) برای تنظیم دکمه مناسب روی true استفاده می‌کنیم؛ به طور مشابه این کار زمانی که یک دکمه آزاد می‌شود اعمال می‌شود (SDL_JOYBUTTONUP). این برای جابجایی دکمه بسیار مشکل است. بیایید در کلاس خود آن را امتحان

کنیم. این تقریبا برای مدیریت دکمه است. بیایید این را در کلاس Player خودمان آزمایش کنیم.

```
if(TheInputHandler::Instance()->getButtonState(0, 3))
{
    m_velocity.setX(1);
}
```

در اینجا ما بررسی می‌کنیم که آیا دکمه 3 فشرده شده است (زرد یا Y روی یک کنترل‌کننده ایکس باکس) و اگر چنین بود سرعت خودمان را تنظیم می‌کنیم. این تمام چیزی است که ما در مورد جوی‌استیک در این کتاب پوشش خواهیم داد. شما درک خواهید کرد که پشتیبانی از بسیاری از جوی‌استیک بسیار مشکل است و نیازمند تغییرات زیادی است تا اطمینان حاصل شود که هر یک به درستی مدیریت می‌شود. با این حال، راه‌هایی وجود دارد که از طریق آن می‌توانید شروع به داشتن پشتیبانی برای بسیاری از جوی‌استیک ها کنید؛ برای مثال، از طریق یک فایل پیکربندی و یا حتی با استفاده از وراثت برای انواع مختلف جوی‌استیک.

مدیریت رویدادهای ماوس

بر خلاف جوی‌استیک ها، ما نباید ماوس را مقداردهی کنیم. همچنین ما می‌توانیم با اطمینان فرض کنیم که فقط یک ماوس در هر زمان متصل می‌شود، بنابراین نیازی نیست که چندین دستگاه ماوس را مدیریت کنیم. ما می‌توانیم با نگاه کردن به رویدادهای در دسترس ماوس که SDL پوشش می‌دهد، شروع کنیم:

رویداد ماوس SDL	هدف
SDL_MouseButtonEvent	یک دکمه روی ماوس فشرده یا رها شده است

SDL_MouseMotionEvent	ماوس جابجا شده است
SDL_MouseWheelEvent	غلتک ماوس حرکت کرده است

درست مانند رویدادهای جوی‌استیک، هر رویداد ماوس یک نوع مقدار دارد، جدول زیر این مقدارها را نشان می‌هد:

رویداد ماوس SDL	نوع مقدار
SDL_MouseButtonEvent	SDLMOUSEBUTTONDOWN یا SDL_MOUSEBUTTONUP
SDL_MouseMotionEvent	SDL_MOUSEMOTION
SDL_MouseWheelEvent	SDL_MOUSEWHEEL

ما هیچ رویداد حرکت غلتک ماوس را پیاده سازی نمی‌کنیم چون اغلب بازی‌ها از آن‌ها استفاده نمی‌کنند.

استفاده از رویدادهای دکمه ماوس

پیاده سازی رویدادهای دکمه ماوس به اندازه رویدادهای جوی‌استیک ساده است، حتی ساده تر چنانکه فقط سه دکمه، چپ، راست و میانی برای انتخاب کردن از آن داریم. SDL اینها را به صورت 0 برای چپ، 1 برای میانی و 2 برای راست شماره گذاری کرده است.در سرآیند InputHandler خودمان بیایید یک آرایه شبیه به دکمه های جوی‌استیک اعلام کنیم، اما این بار یک آرایه یک بعدی، چنانکه نمی‌خواهیم چند دستگاه ماوس را مدیریت کنیم.

```
std::vector<bool> m_mouseButtonStates;
```

سپس در سازنده InputHandler خودمان می‌توانیم سه وضعیت دکمه ماوس(پیش فرض false) را در آرایه قرار دهیم:

```
for(int i = 0; i < 3; i++)
{
    m_mouseButtonStates.push_back(false);
}
```

به فایل سرآیند خود برگردیم، اجازه دهید یک صفت enum برای کمک به ما با مقادیر دکمه های ماوس ایجاد کنیم. این را بالاتر از کلاس قرار دهید تا فایل‌های دیگری که سرآیند InputHandler را می‌گنجانند نیز بتوانند از آن استفاده کنند.

```
enum mouse_buttons
{
    LEFT = 0,
    MIDDLE = 1,
    RIGHT = 2
};
```

حالا بیایید رویدادهای ماوس در حلقه رویداد را مدیریت کنیم:

```
 if(event.type == SDL_MOUSEBUTTONDOWN)
{
    if(event.button.button == SDL_BUTTON_LEFT)
    {
        m_mouseButtonStates[LEFT] = true;
    }

    if(event.button.button == SDL_BUTTON_MIDDLE)
    {
        m_mouseButtonStates[MIDDLE] = true;
    }

    if(event.button.button == SDL_BUTTON_RIGHT)
    {
       m_mouseButtonStates[RIGHT] = true;
    }
}
```

```
if(event.type == SDL_MOUSEBUTTONUP)
{
   if(event.button.button == SDL_BUTTON_LEFT)
   {
      m_mouseButtonStates[LEFT] = false;
   }

   if(event.button.button == SDL_BUTTON_MIDDLE)
   {
      m_mouseButtonStates[MIDDLE] = false;
   }

   if(event.button.button == SDL_BUTTON_RIGHT)
   {
      m_mouseButtonStates[RIGHT] = false;
   }
}
```

ما همچنین به یک تابع برای دسترسی به وضعیت های دکمه ماوس نیاز داریم. بیایید این تابع عمومی را به فایل سرآیند InputHandler اضافه کنیم:

```
bool getMouseButtonState(int buttonNumber)
{
   return m_mouseButtonStates[buttonNumber];
}
```

این تمام چیزی است ما برای رویدادهای دکمه ماوس نیاز داریم. ما اکنون می‌توانیم این را در کلاس Player آزمایش کنیم.

```
if(TheInputHandler::Instance()->getMouseButtonState(LEFT))
{
   m_velocity.setX(1);
}
```

مدیریت رویدادهای حرکت ماوس

رویدادهای حرکت ماوس، به خصوص در عناوین سه‌بعدی اکشن اول شخص یا سوم‌شخص بسیار مهم هستند. برای بازی‌های دو بعدی ما ممکن است بخواهیم کاراکتر ما ماوس را به عنوان راهی برای کنترل شی هایمان دنبال کند، یا ممکن است بخواهیم شی ها به جایی که ماوس کلیک شده بروند (شاید برای یک بازی استراتژی). حتی ممکن است فقط بخواهیم بدانیم که ماوس کجا کلیک شده است تا بتوانیم از آن برای منوها استفاده کنیم. خوشبختانه برای ما، رویدادهای حرکت ماوس نسبتا ساده هستند. ما با ایجاد یک Vector2D* خصوصی در فایل سرآیند شروع خواهیم کرد تا به عنوان متغیر موقعیت برای ماوس خود استفاده کنیم:

```
Vector2D* m_mousePosition;
```

سپس، برای این به یک دستیابی کننده(accessor) عمومی نیاز داریم:

```
Vector2D* getMousePosition()
{
   return m_mousePosition;
}
```

و اکنون می‌توانیم این را در حلقه رویداد خود مدیریت کنیم:

```
if(event.type == SDL_MOUSEMOTION)
{
   m_mousePosition->setX(event.motion.x);
   m_mousePosition->setY(event.motion.y);
}
```

این تمام چیزی است که برای حرکت ماوس نیاز داریم. پس بیایید Player را کاری کنیم تا موقعیت ماوس را دنبال کند تا این ویژگی را آزمایش کنیم:

```
Vector2D* vec = TheInputHandler::Instance()->getMousePosition();

m_velocity = (*vec - m_position) / 100;
```

در اینجا ما سرعت خود را روی برداری از موقعیت فعلی بازیکن به موقعیت ماوس تنظیم کرده‌ایم. شما می‌توانید این بردار را با کم کردن مکان مورد نظر از مکان فعلی به دست آورید؛ ما از قبل یک بردار با عملگر سربارگذاری شده منها را داریم که این کار برای ما آسان می‌کند. ما همچنین بردار را بر ۱۰۰ تقسیم می‌کنیم؛ این فقط اندکی سرعت را کم می‌کند به طوری که ما می‌توانیم آن را به جای چسبیدن به موقعیت ماوس، دنبال کننده ببینیم. / را بردارید تا شی شما ماوس را دقیقا دنبال کند.

پیاده سازی ورودی صفحه‌کلید

روش نهایی ما برای ورودی و ساده‌ترین بین آن‌ها ورودی صفحه‌کلید است. ما نباید هیچ رویداد حرکت را مدیریت کنیم، ما فقط می‌خواهیم وضعیت هر دکمه را ببینیم. ما نمی‌خواهیم در اینجا یک آرایه را اعلام کنیم زیرا SDL یک تابع درون-ساخت دارد که به ما یک آرایه با وضعیت هر کلید می‌دهد؛ 1 فشرده شده است و 0 فشرده نشده است.

```
SDL_GetKeyboardState(int* numkeys)
```

پارامتر numkeys تعداد کلیدهای در دسترس روی صفحه‌کلید (طول آرایه keystate) را باز خواهند گرداند. بنابراین در فایل سرآیند InputHandler ما می‌توانیم یک اشاره گر به آرایه‌ای اعلام کنیم که از SDL_GetKeyboardState بازگردانده می‌شود.

```
Uint8* m_keystates;
```

زمانی که مدیریت کننده رویداد خود را به روز رسانی می‌کنیم، همچنین می‌توانیم وضعیت کلیدها را به روزرسانی کنیم؛ این را در بالای حلقه رویداد خود قرار دهید.

```
m_keystates = SDL_GetKeyboardState(0);
```

در حال حاضر نیاز به ایجاد یک تابع ساده داریم که بررسی می‌کند آیا یک کلید پایین است یا خیر.

```
bool InputHandler::isKeyDown(SDL_Scancode key)
{
   if(m_keystates != 0)
   {
      if(m_keystates[key] == 1)
      {
        return true;
      }
      else
      {
         return false;
      }
   }

   return false;
}
```

این تابع SDL_SCANCODE را به عنوان یک پارامتر می‌گیرد. فهرست کامل مقادیر SDL_SCANCODE را می توان در مستندات SDL در http://wiki.libsdl.org/moin.cgi یافت.

ما می‌توانیم کلیدها را در کلاس Player آزمایش کنیم. ما از کلیدهای جهت دار(arrow keys) برای حرکت بازیکن(player) استفاده خواهیم کرد.

```
if(TheInputHandler::Instance()->isKeyDown(SDL_SCANCODE_RIGHT))
{
   m_velocity.setX(2);
}

if(TheInputHandler::Instance()->isKeyDown(SDL_SCANCODE_LEFT))
{
   m_velocity.setX(-2);
```

```
}

if(TheInputHandler::Instance()->isKeyDown(SDL_SCANCODE_UP))
{
   m_velocity.setY(-2);
}
if(TheInputHandler::Instance()->isKeyDown(SDL_SCANCODE_DOWN))
{
   m_velocity.setY(2);
}
```

اکنون ما به درستی مدیریت کننده کلید را داریم. هر تعداد کلید که می‌توانید را آزمایش کنید و SDL_Scancode را برای کلیدهایی که به احتمال زیاد می‌خواهید از آن‌ها استفاده کنید، را بررسی کنید.

بسته‌بندی چیزها

ما اکنون همه دستگاه هایی را که قرار است مدیریت کنیم را پیاده سازی کرده‌ایم، اما در حال حاضر حلقه رویداد ما کمی آشفته است. ما باید آن را به بخش‌های کنترل پذیر تری تقسیم کنیم. ما این کار را با استفاده از یک دستور switch برای انواع رویداد و چند تابع خصوصی در InputHandler انجام خواهیم داد. ابتدا اجازه دهید توابع خود را در فایل سرآیند اعلام کنیم:

```
// private functions to handle different event types

// handle keyboard events
void onKeyDown();
void onKeyUp();

// handle mouse events
void onMouseMove(SDL_Event& event);
void onMouseButtonDown(SDL_Event& event);
void onMouseButtonUp(SDL_Event& event);
```

```
// handle joysticks events
void onJoystickAxisMove(SDL_Event& event);
void onJoystickButtonDown(SDL_Event& event);
void onJoystickButtonUp(SDL_Event& event);
```

ما رویداد را از حلقه رویداد به هر تابع (غیر از کلیدها) ارسال می‌کنیم تا بتوانیم بر این اساس آن‌ها را کنترل کنیم. اکنون باید دستور switch مان را در حلقه رویداد ایجاد کنیم.

```
void InputHandler::update()
{
   SDL_Event event;
   while(SDL_PollEvent(&event))
   {
      switch (event.type)
     {
      case SDL_QUIT:
         TheGame::Instance()->quit();
      break;
      case SDL_JOYAXISMOTION:
         onJoystickAxisMove(event);
       break;
       case SDL_JOYBUTTONDOWN:
          onJoystickButtonDown(event);
       break;
       case SDL_JOYBUTTONUP:
          onJoystickButtonUp(event);
       break;
       case SDL_MOUSEMOTION:
          onMouseMove(event);
       break;
       case SDL_MOUSEBUTTONDOWN:
          onMouseButtonDown(event);
       break;
       case SDL_MOUSEBUTTONUP:
```

```
            onMouseButtonUp(event);
        break;
        case SDL_KEYDOWN:
          onKeyDown();
        break;
        case SDL_KEYUP:
            onKeyUp();
        break;
        default:
        break;
        }
    }
}
```

همانطور که می‌توانید ببینید، ما اکنون حلقه رویداد را از هم جدا کرده ایم و تابع مربوطه(associated) را بسته به نوع رویداد فراخوانی می‌کنیم. ما اکنون می‌توانیم تمام کارهای قبلی مان را بین این توابع تقسیم کنیم؛ برای مثال، ما می‌توانیم مدیریت فشرده شدن تمام دکمه ماوس را در تابع onMouseButtonDown قرار دهیم.

```
void InputHandler::onMouseButtonDown(SDL_Event& event)
{
    if(event.button.button == SDL_BUTTON_LEFT)
    {
        m_mouseButtonStates[LEFT] = true;
    }

    if(event.button.button == SDL_BUTTON_MIDDLE)
    {
        m_mouseButtonStates[MIDDLE] = true;
    }

    if(event.button.button == SDL_BUTTON_RIGHT)
    {
        m_mouseButtonStates[RIGHT] = true;
```

```
    }
}
```

بقیه کدها برای InputHandler در داخل دانلودهای کد منبع در دسترس هستند.

خلاصه

ما چند مطلب پیچیده را در این بخش پوشش داده‌ایم. ما به مقدار کمی از ریاضیات برداری نگاه کردیم و اینکه چگونه می‌توانیم از آن برای حرکت شی های بازی خود استفاده کنیم. ما همچنین مقدار دهی و استفاده از چند جوی‌استیک و محور و استفاده از ماوس و صفحه‌کلید را نیز پوشش داده‌ایم. بالاخره، همه چیز را با یک روش تمیز برای مدیریت کردن رویدادهای خود بسته‌بندی کردیم.