

PainterEngine 快速入门教程

Author:DBinary

[Matrixcascade@gmail.com](mailto:Matrixcascade@gmail.com)

## PainterEngine导引

PainterEngine是一个基于C语言编写的完全开源的图形游戏引擎。

PainterEngine完全独立于操作系统、编译器、工具链及运行时环境，这意味着PainterEngine可以被移植到任意提供C语言编译环境的平台中，包括但不限windows，linux，android，ios及一系列单片机裸机环境中。

PainterEngine的设计理念精简，学习曲线平缓，你只需要花很少的时间在引擎的学习上，耗费极少的学习时间成本，快速实现你需要的功能。

## PainterEngine 目录结构

\*Core：PainterEngine基础算法库

\*Kernel：PainterEngine模块代码库（基于Core）

Architecture：PainterEngine功能代码库（基于Kernel）

Platform：平台兼容层代码库

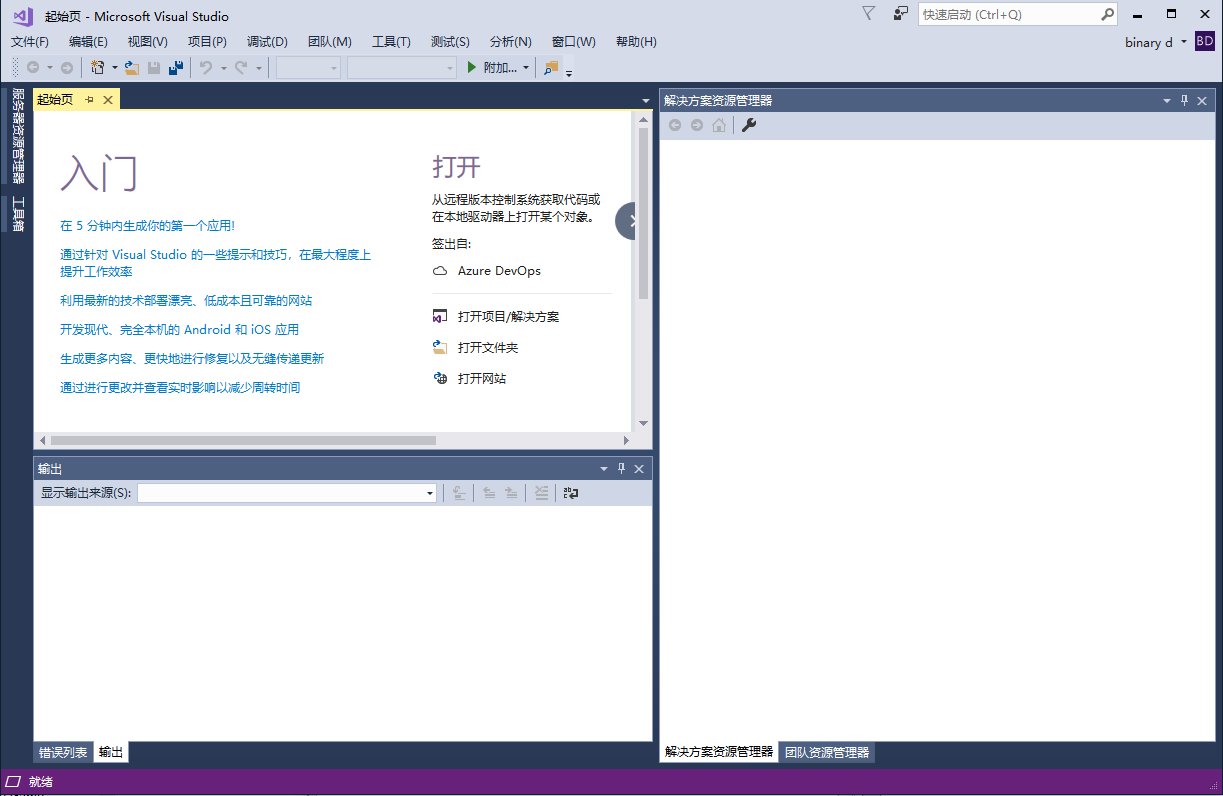
Support：相关支持、帮助文档

## PainterEngine开发环境

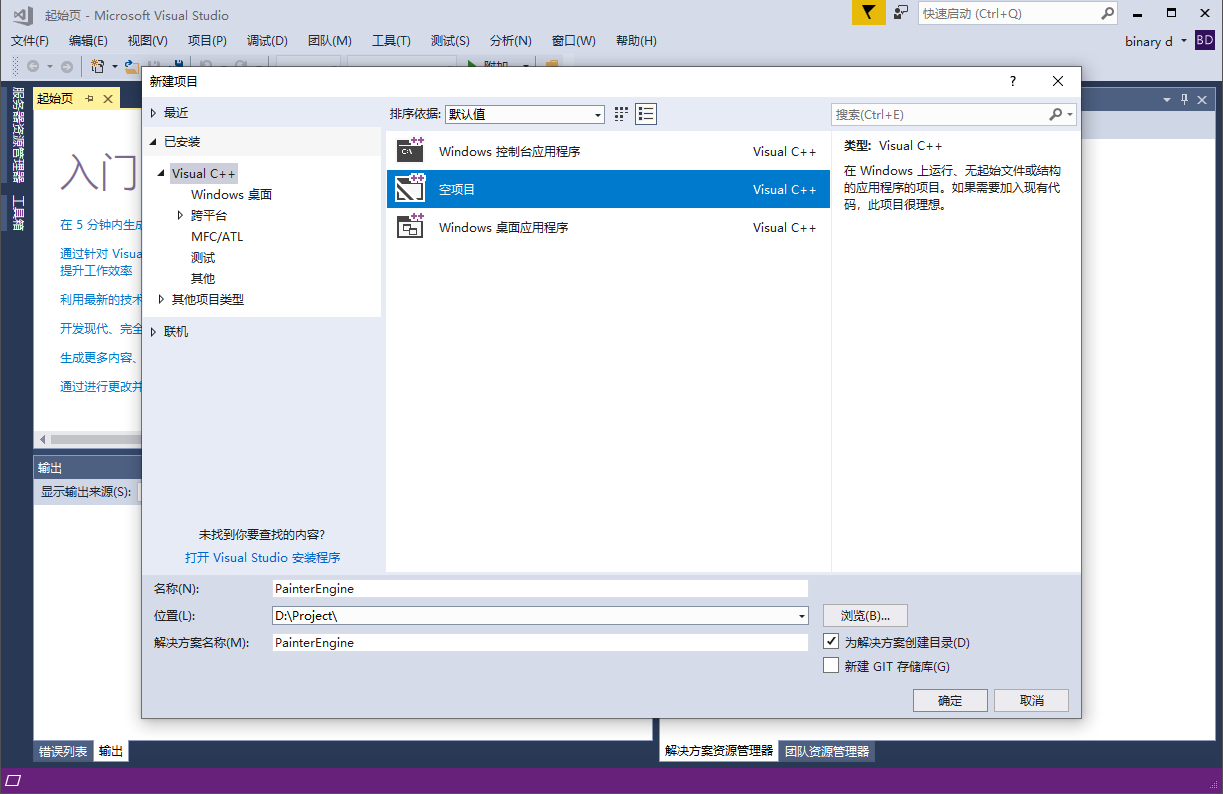
### Windows

##### Visual Studio

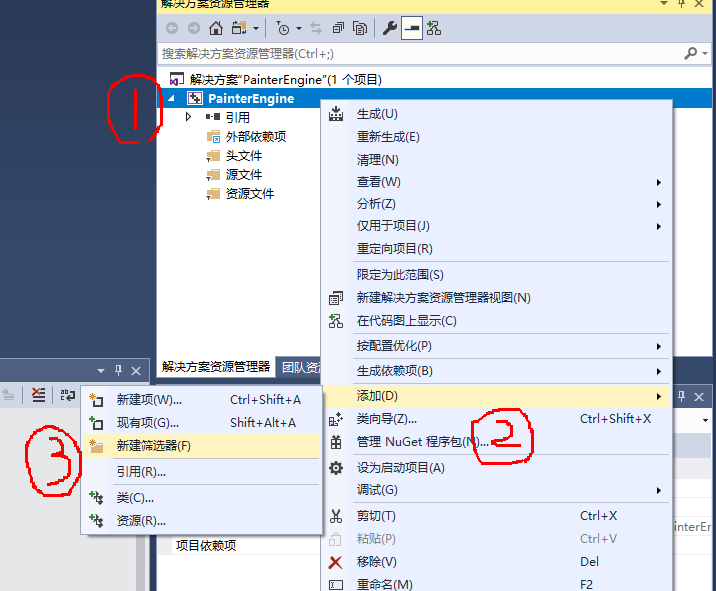
1.启动visual studio



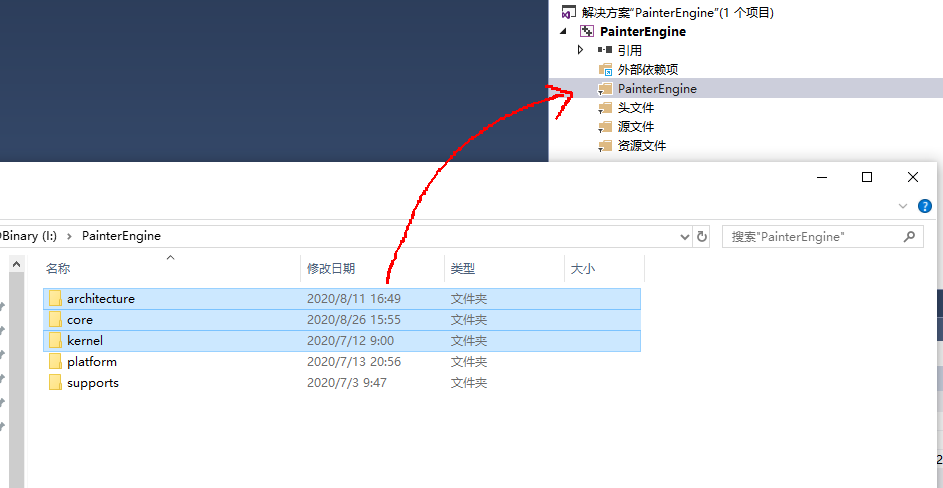
2.菜单--文件—新建—项目—创建一个空项目



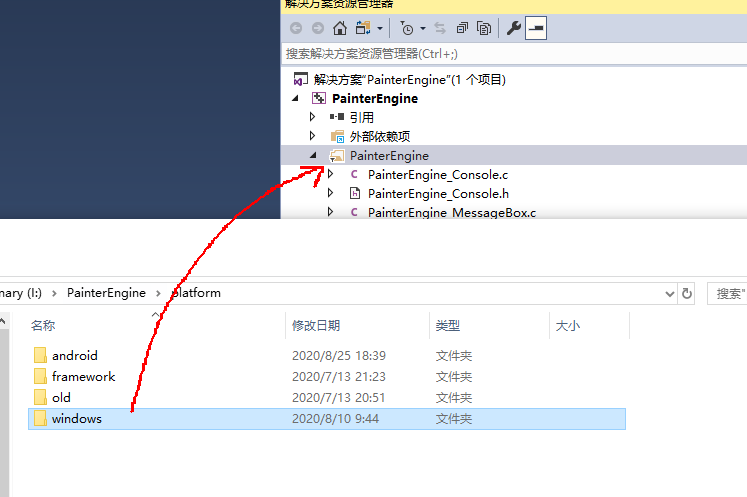
3.在项目上右键---添加---新建筛选器—命名为PainterEngine（非必须步骤，可跳过）



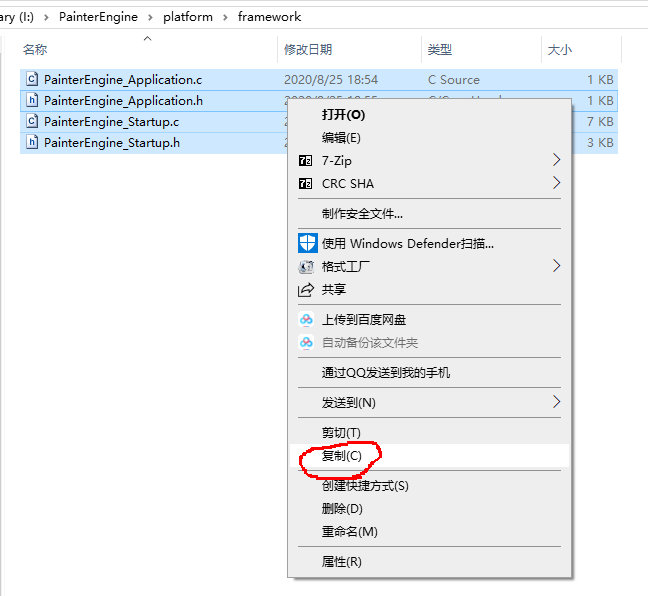
4.将PainterEngine目录下kernel、core、architecture三个文件夹（所有文件拖到新建的筛选器中，如跳过了步骤3可直接拖到项目中）



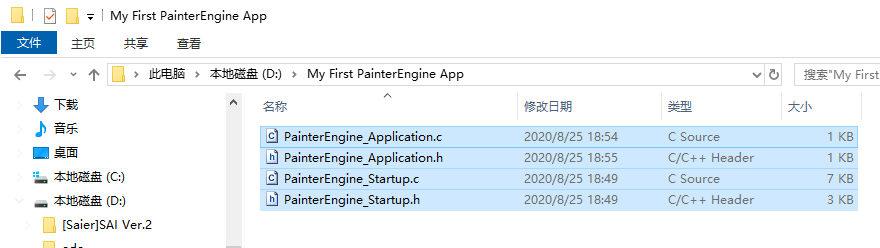
5.进入PainterEngine/platform,因为我们开发的是windows程序,因此将windows目录的同样拖动到筛选器中。



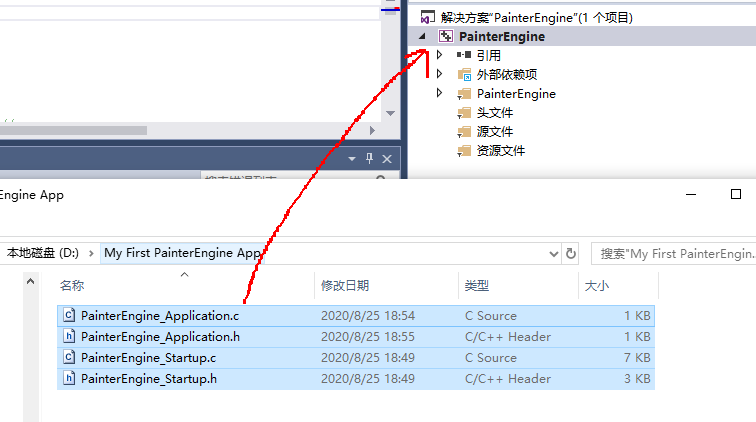
6.进入PainterEngine/platform/framework,复制当中的所有文件



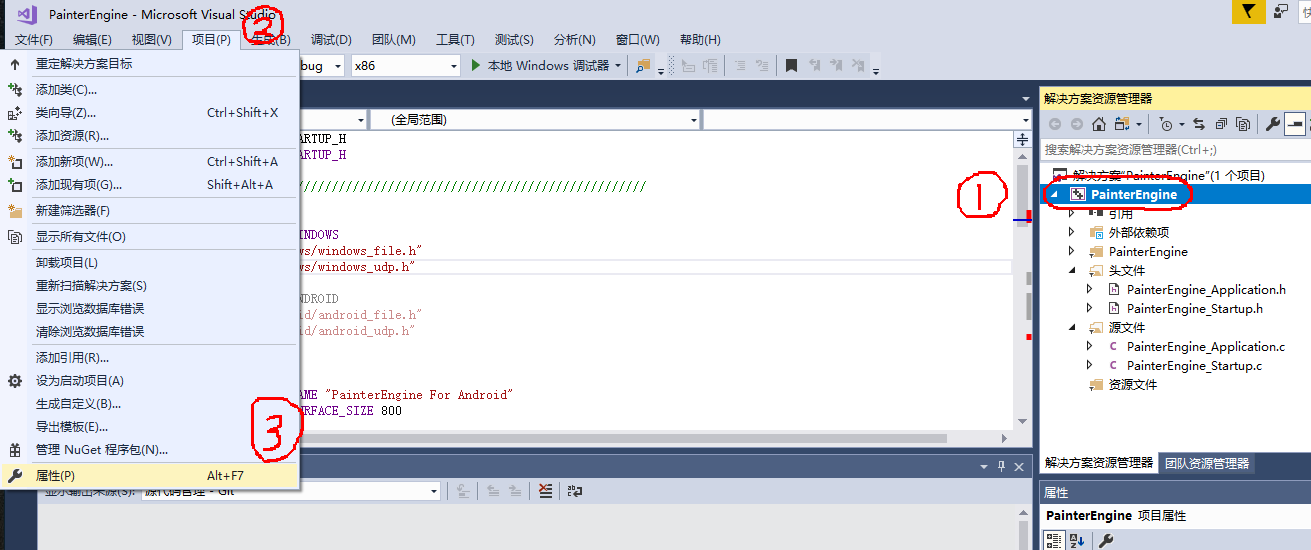
7.挑一个你喜欢的地方,新建一个文件夹(作为你的工程代码目录),将上面的文件复制到这个文件夹中(例如D盘中我新建了一个My First PainterEngine Project作为示范)



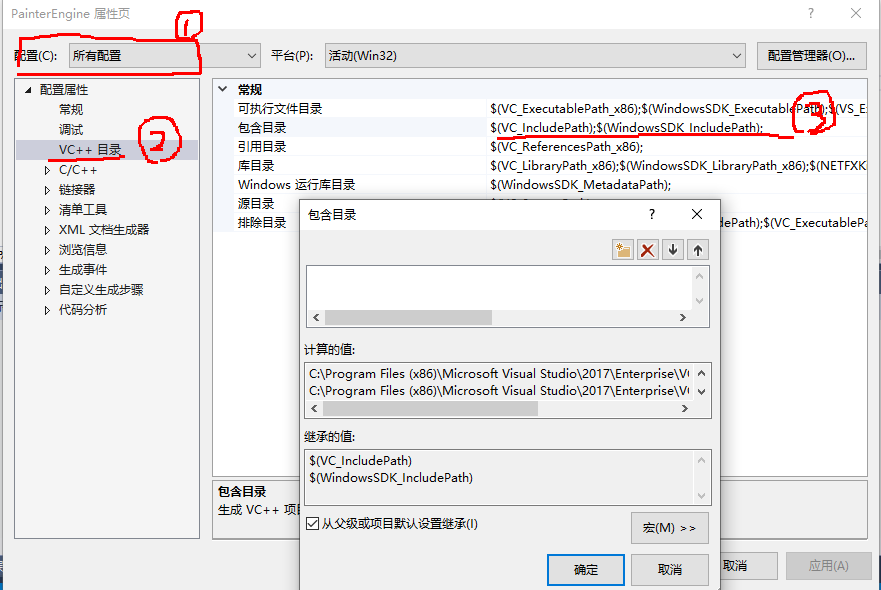
8.将复制的文件黏贴到这个文件夹中,并将这些文件拖动到项目中



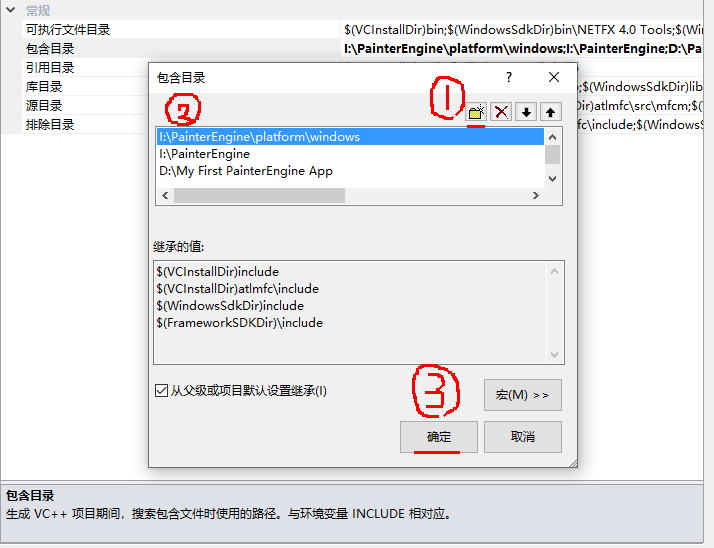
9.选中项目,点击菜单项目—属性



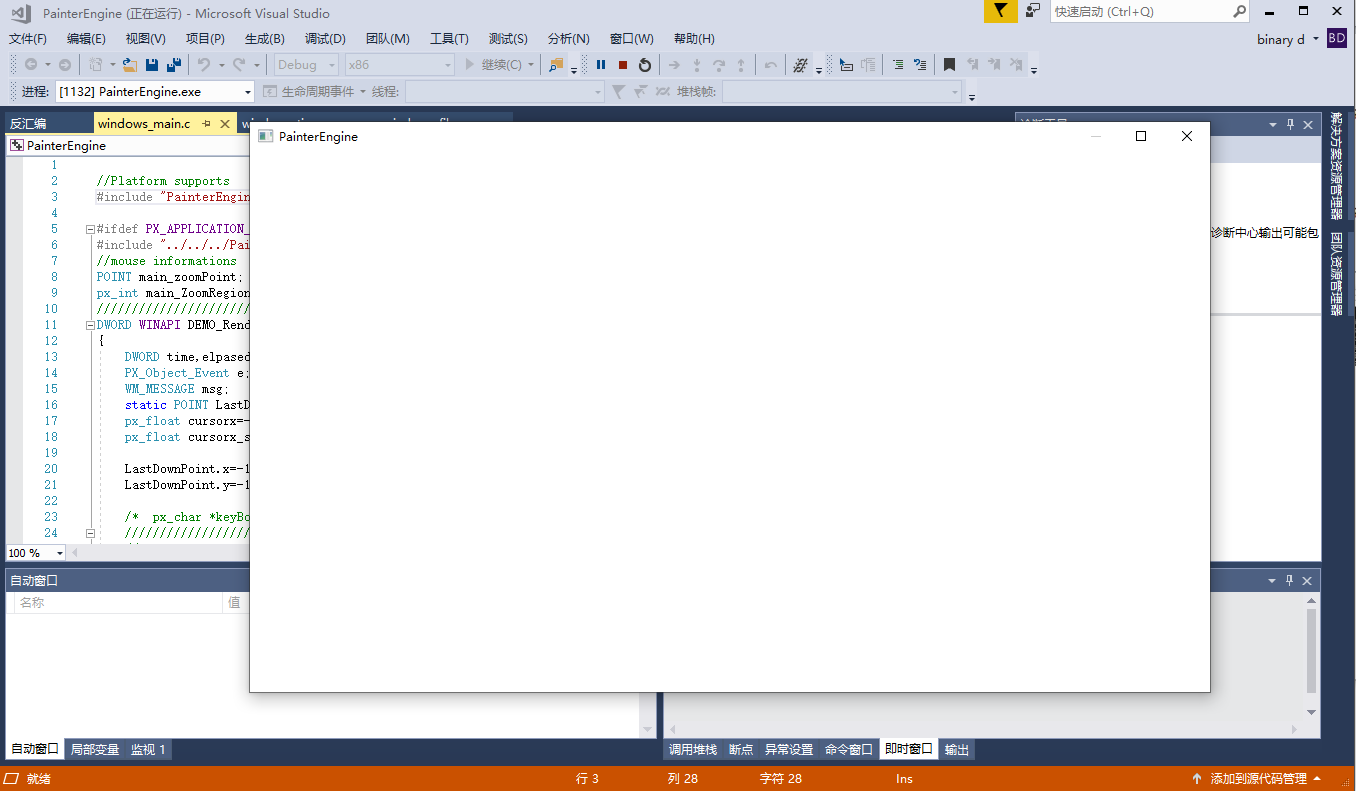
10.将配置改成所有配置---VC++目录---包含目录



11.将PainterEngine的所在目录,windows平台库目录和你工程代码目录添加进来,然后点击确定



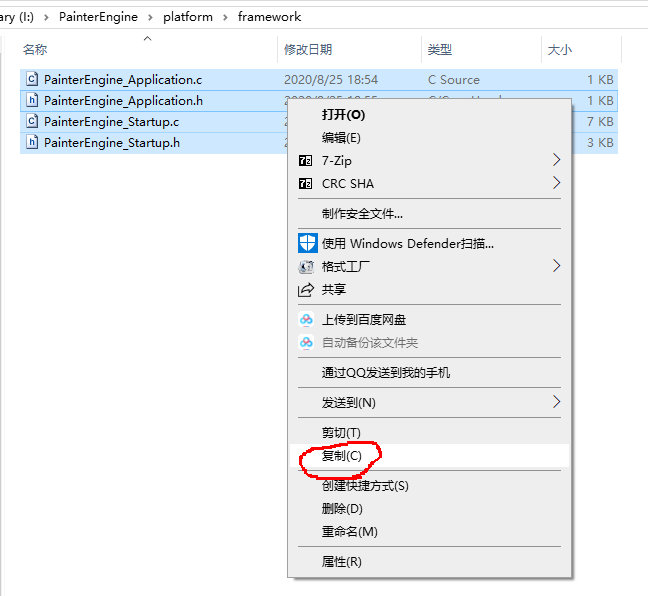
12.现在,你可以试运行第一个PainterEngine项目了



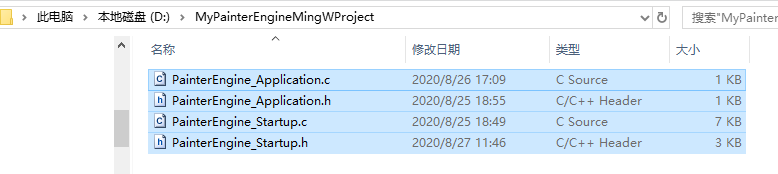
##### MingW(CLoin,Visual Studio code)

1.下载并安装mingw,设置环境变量(请参照mingw相关教程,过程略).

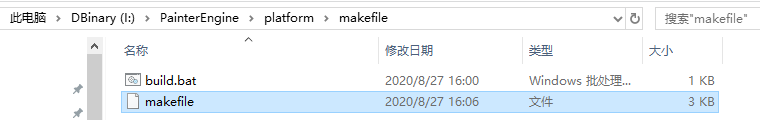
进入PainterEngine/platform/framework,复制当中的所有文件

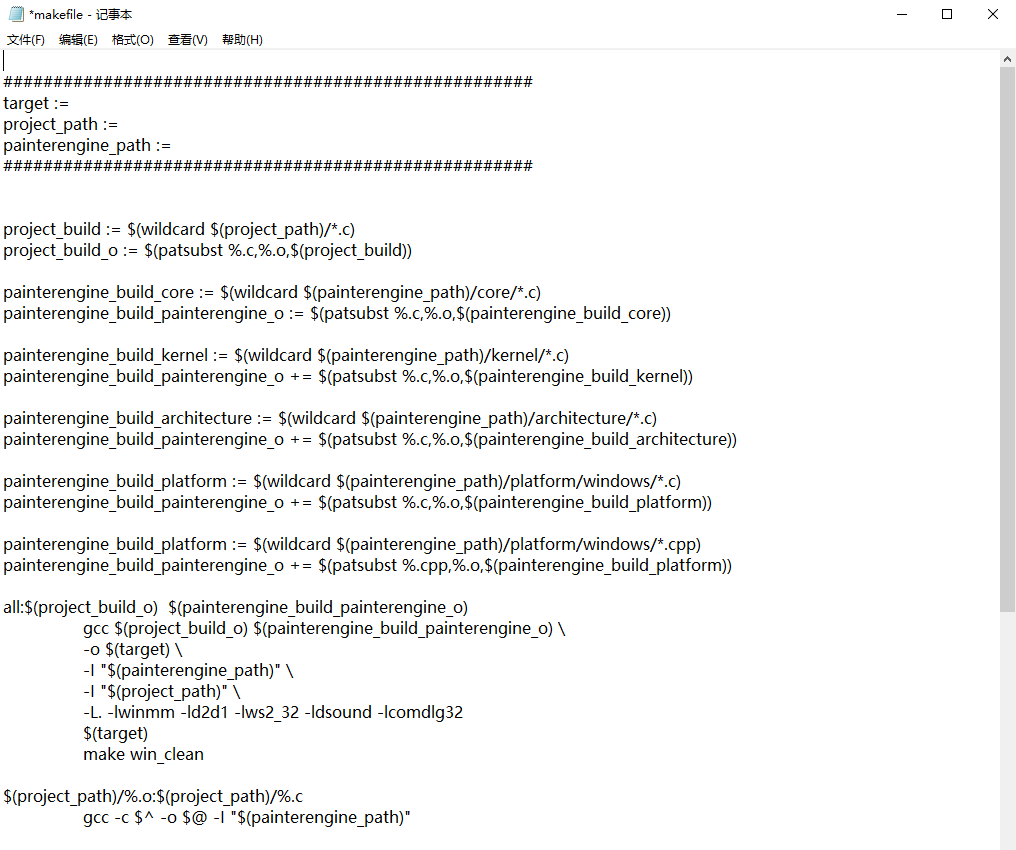


2.挑一个你喜欢的地方,新建一个文件夹(作为你的工程代码目录),将上面的文件复制到这个文件夹中(例如D盘中我新建了一个MyPainterEngineMingWProject作为示范,**注意文件夹名不要带有空格**)



3.进入PainterEngine\platform\makefile目录,用记事本打开makefile文件





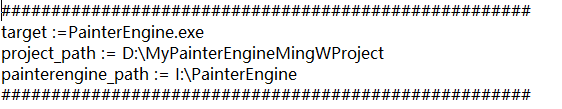
4.修改makefile的配置,包含生成exe的路径,工程代码的路径和PainterEngine的路径

其中target为生成exe的路径

project\_path为工程代码的路径

painterengine\_path 为PainterEngine路径

**(以上路径注意不要带有空格)**



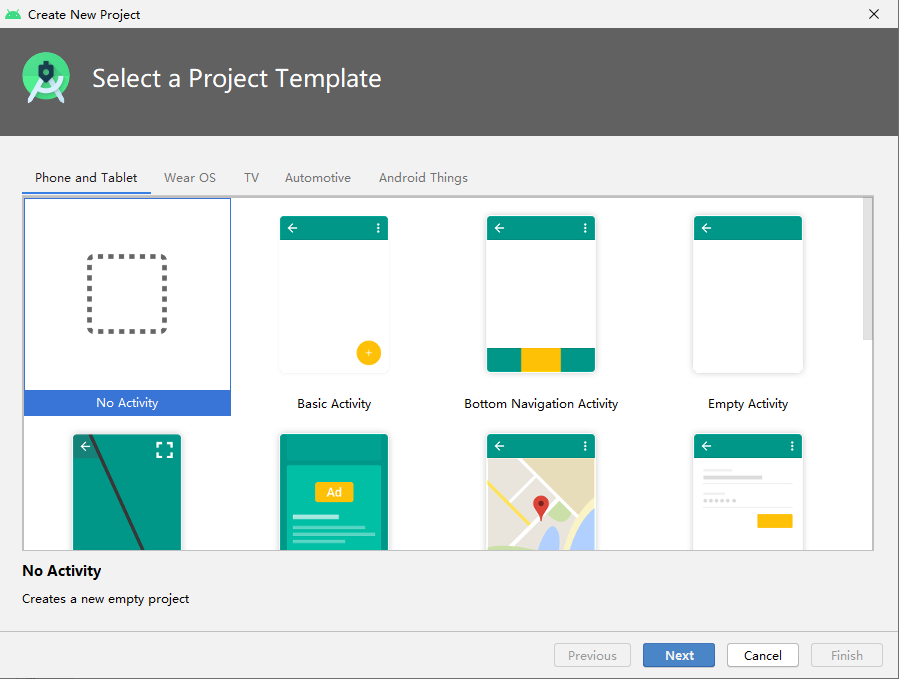
5.双击PainterEngine\platform\makefile\build.bat,编译运行



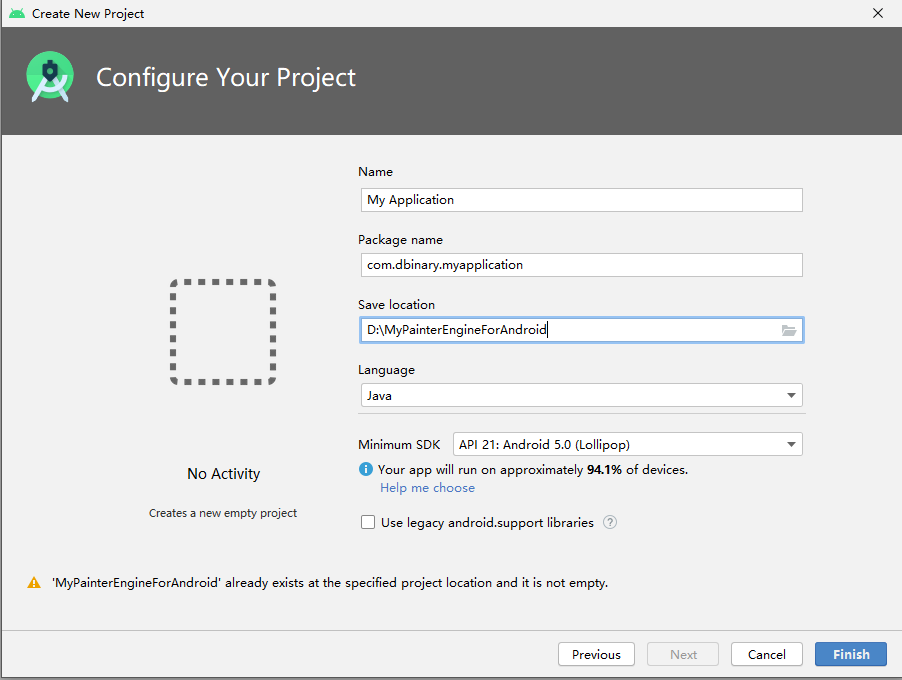
### Android

##### Android studio

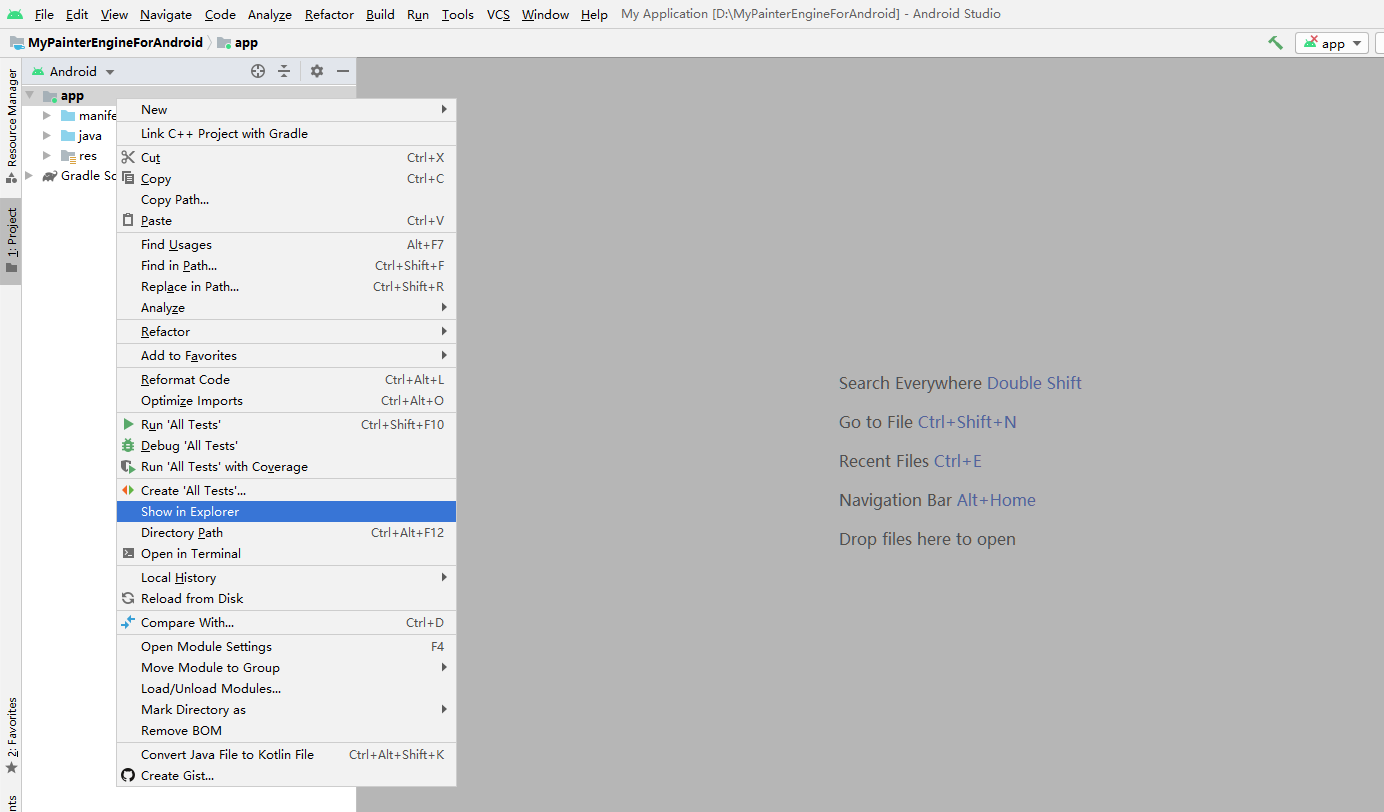
1.打开android studio,File---new—new project—选择No Activity然后点击Next



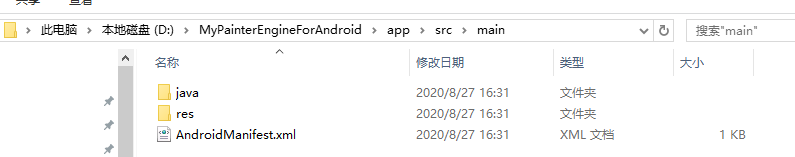
2.选择一个你喜欢的位置,创建项目文件(以D:\MyPainterEngineForAndroid为例),点击finish



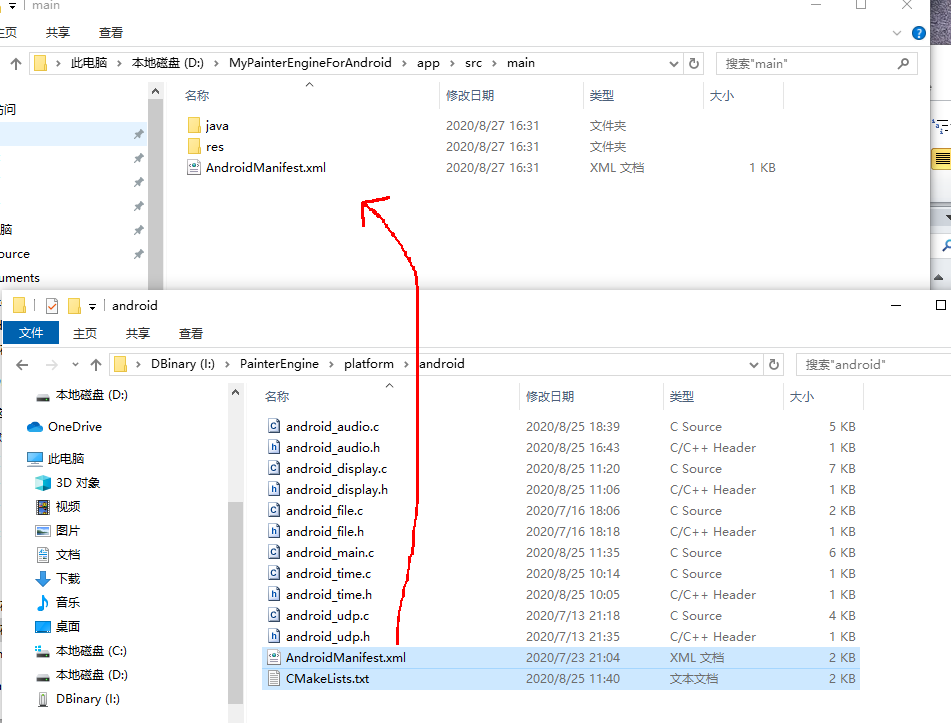
3.在项目目录的app中点击右键,选择show in explorer



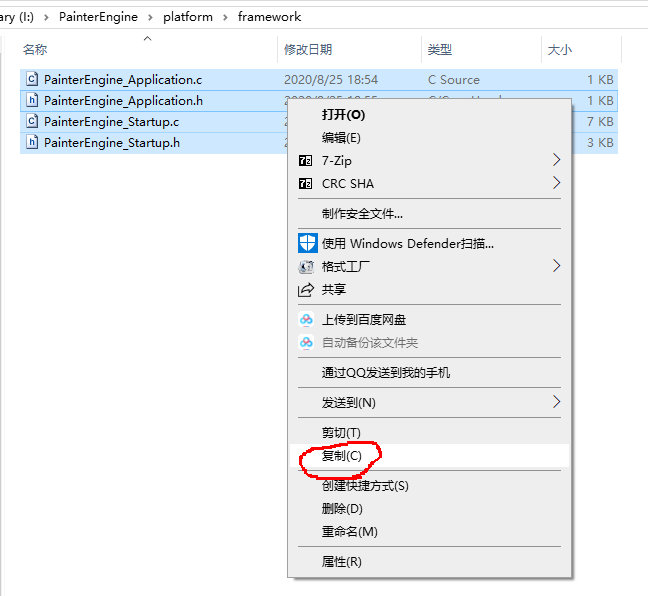
4.依次进入\app\src\main



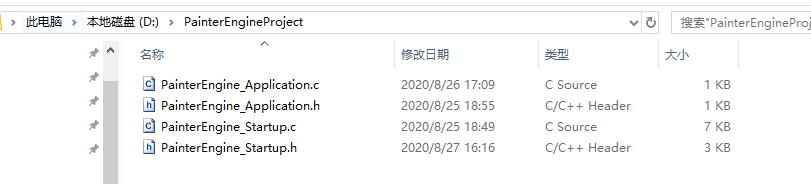
5.打开PainterEngine目录下的platform/android,将AndroidManifest.xml和CMakeList覆盖和复制到\app\src\main



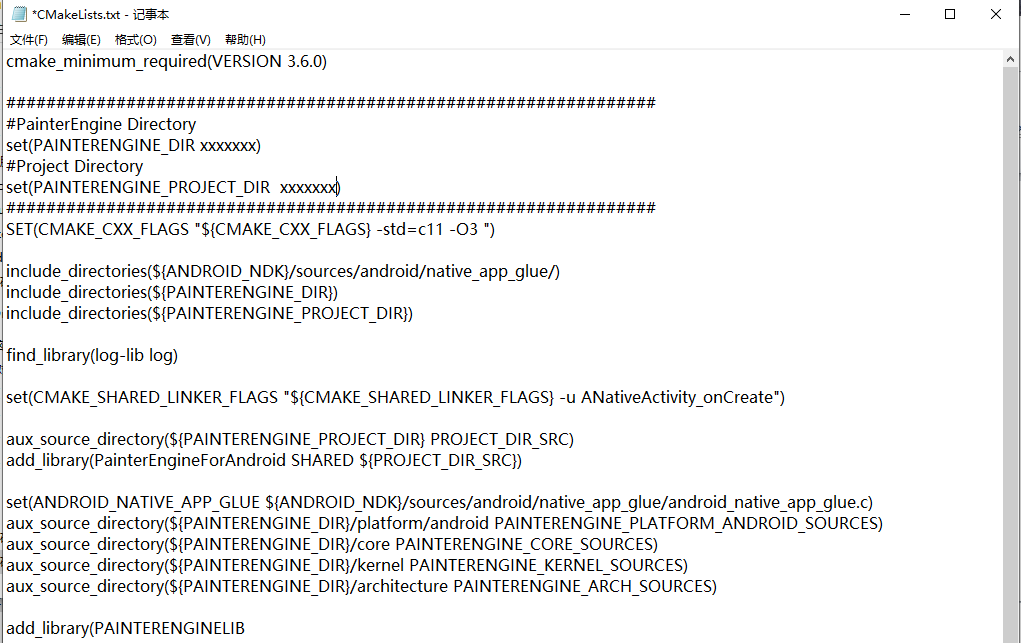
5. 进入PainterEngine/platform/framework,复制当中的所有文件



6.挑一个你喜欢的地方,新建一个文件夹(作为你的工程代码目录),将上面的文件复制到这个文件夹中(例如D盘中我新建了一个PainterEngineProject作为示范,**注意文件夹名不要带有空格**)



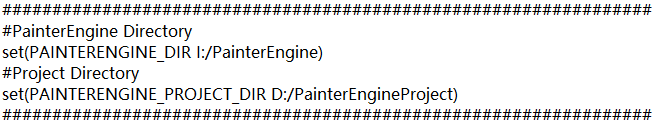
7.打开\app\src\main\CMakeLists.txt,



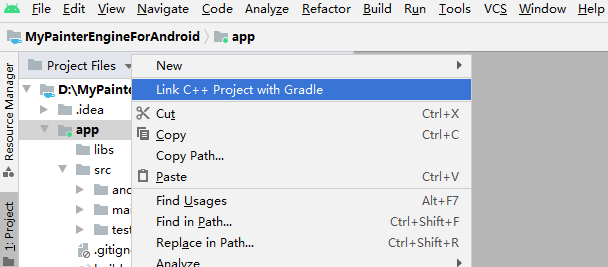
将

PAINTERENGINE\_DIR 修改为你的PainterEngine路径

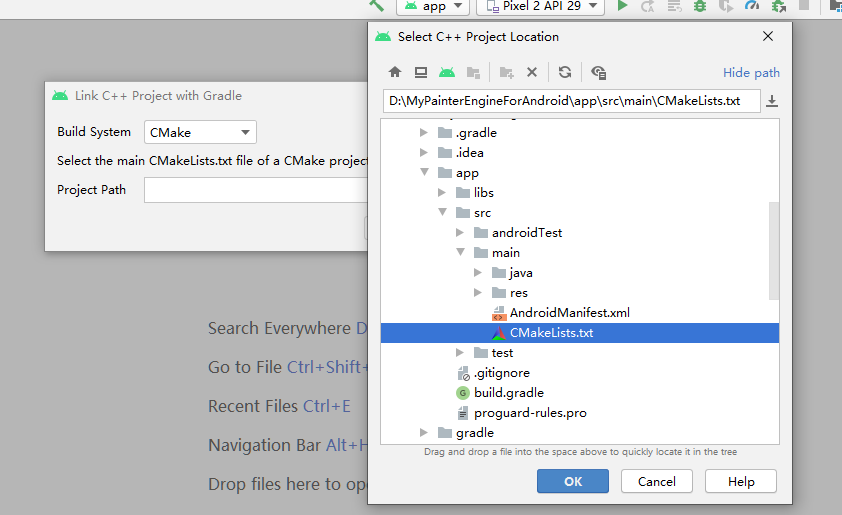
PAINTERENGINE\_PROJECT\_DIR 修改为你的工程代码目录路径

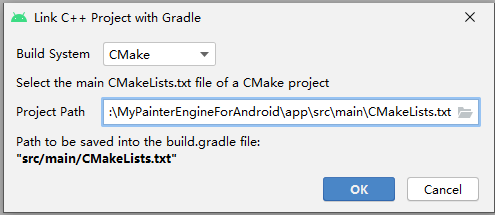


8.回到Android Studio,选中App,右键点击Link C++ Project With Gradle

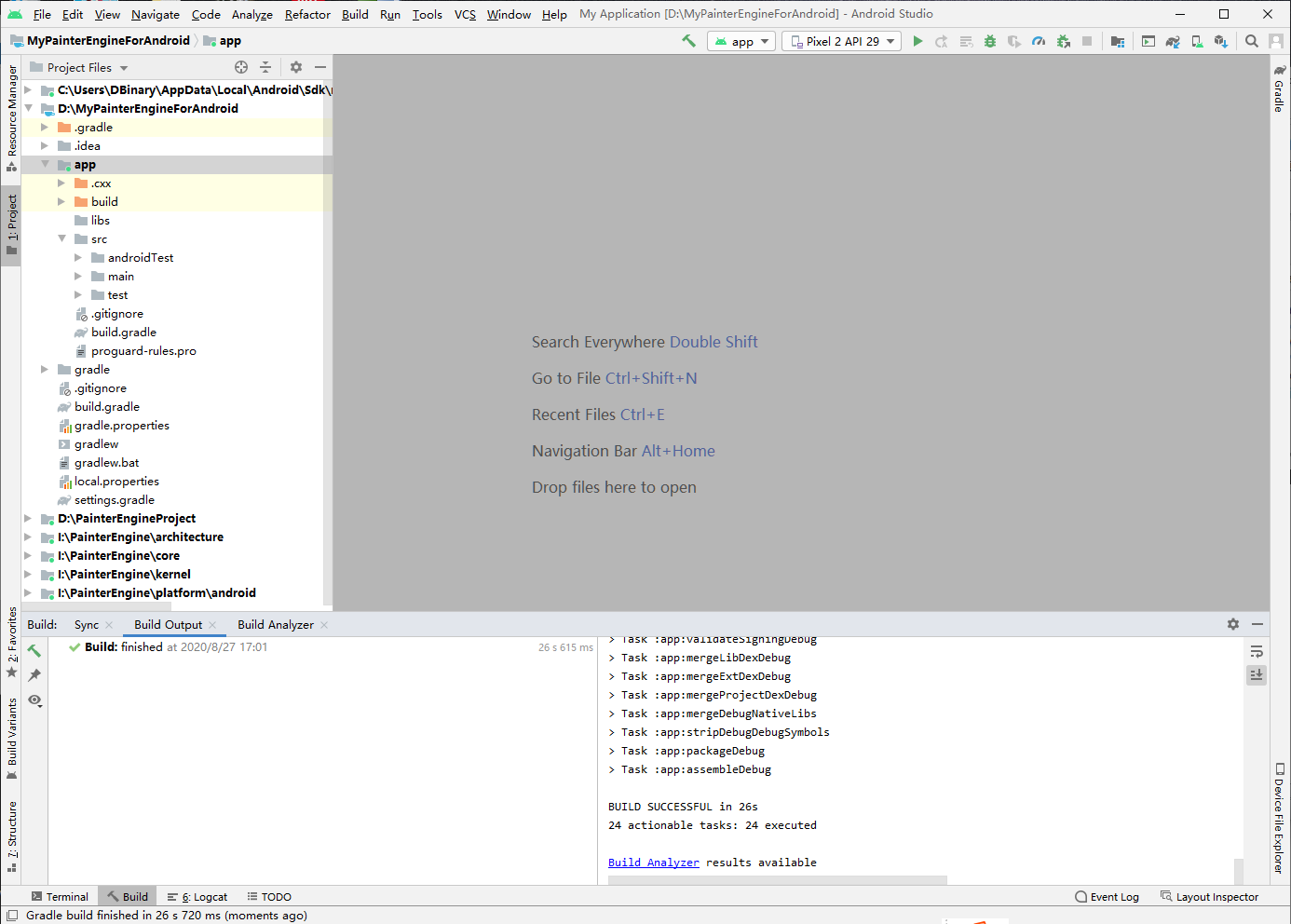


9.选择之前配置好的CMakeLists.txt





10.在Android Studio中按下Ctrl+F9或点击菜单栏上的build



编译成功,现在你可以连接手机运行PainterEngine的Android应用了



## PainterEngine基础框架结构

### Application 基础架构

PainterEngine architecture包含着PainterEngine默认的运行时配置与执行框架,使用PainterEngine的框架体系开发PainterEngine程序是强烈建议的.

别担心,PainterEngine的框架极为简单,PainterEngine将占用您短短几分钟的时间快速搭建并了解其框架。

打开工程目录

PainterEngine\_Startup PainterEngine运行时框架,在当中定义了内存池大小,运行平台,窗体大小等应用的基本参数.

PainterEngine\_Application 为功能逻辑运行框架,是编写用户代码的主要地方.

打开PainterEngine\_Application.c,你可以看到三个函数

#### PX\_ApplicationInitialize

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

PainterEngine初始化函数,在该函数中,完成应用的一系列初始化操作,例如,在这个函数中将会先初始化PainterEngine的运行时环境，创建的PainterEngine应用的窗口，或者如果你有对音频播放的需求你可以你加上对音频的支持。

其中

pApp 是应用的描述结构

Screen\_width 是当前屏幕的像素宽度

Screen\_height是当前屏幕的像素高度

当然为了减少PainterEngine部署难度，你可以看到在这个函数已经调用了一个默认的初始化函数

PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

这个函数将初始化一个最初默认的运行时环境，并依据当前的屏幕创建了一个窗口。你可以在PainterEngine\_Startup.c找到这个函数的详细实现,但在入门阶段和之后开发应用的大部分情况，你都可以直接使用这个默认的初始化函数或对其进行少量的修改,这个部分将在之后的章节详细介绍。

#### PX\_ApplicationUpdate

px\_void PX\_ApplicationUpdate(PX\_Application \*pApp,px\_dword elpased)

PainterEngine的更新函数,在大部分的情况下,PainterEngine的运行环境会循环调用这个函数。

其中

pApp 是应用的描述结构

elapsed 是上次更新后经历的时间，单位是毫秒

对于其详细应用将在之后详细介绍

#### PX\_ApplicationRender

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

PainterEngine的渲染函数,在大部分的情况下,PainterEngine的运行环境会循环调用这个函数（实时渲染），这个函数的主要作用是绘制图像，当然，为了让显示和动画看起来尽可能流畅，你应该保证这个函数在每秒钟能被执行60次以上。但如果你对渲染没有非常高的实时性要求，你可以降低其帧率以实现更低的渲染功耗。

在这个函数中，你可以看到两行已经写好的代码

px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

其中，第一句中的pRenderSurface表示渲染表面

PX\_RuntimeRenderClear表示用某种颜色清空当前的渲染表面（或者绘制背景色），在上面PX\_COLOR是一个颜色生成函数，四个参数分别表示a，r，g，b颜色分量

简单来说，PainterEngine的绘制流程是，先将surface成背景色，然后在上面绘制图像，在下一帧，再将这个surface重新刷成背景色，再画下一帧的图像。

其中

pApp 是应用的描述结构

elapsed 是上次更新后经历的时间，单位是毫秒

对于其详细应用将在之后详细介绍

#### PX\_ApplicationPostEvent

这个函数用户响应用户输入事件,最常见的是鼠标移动,键盘按键,触摸屏按键等事件

px\_void PX\_ApplicationPostEvent(PX\_Application \*pApp,PX\_Object\_Event e)

其中:

pApp是应用的描述结构

e是事件类型,

e.event有以下几种常用事件类型

#define PX\_OBJECT\_EVENT\_ANY                 0 //任意类型

#define PX\_OBJECT\_EVENT\_CURSORMOVE          1 //光标移动

#define PX\_OBJECT\_EVENT\_CURSORUP            2 //光标抬起

#define PX\_OBJECT\_EVENT\_CURSORRDOWN         3 //光有右键按下

#define PX\_OBJECT\_EVENT\_CURSORDOWN          4 //光标按下(触摸屏按下)

#define PX\_OBJECT\_EVENT\_CURSORRUP           5 //光标抬起(触摸屏抬起)

#define PX\_OBJECT\_EVENT\_CURSOROVER          6 //光标移动到区域\*

#define PX\_OBJECT\_EVENT\_CURSOROUT           7 //光标移出区域\*

#define PX\_OBJECT\_EVENT\_CURSORWHEEL         8 //光标滚轮\*

#define PX\_OBJECT\_EVENT\_STRING              9 //输入法字符串\*

#define PX\_OBJECT\_EVENT\_EXECUTE             10 //按钮按下

#define PX\_OBJECT\_EVENT\_CURSORCLICK         11 //光标单击\*

#define PX\_OBJECT\_EVENT\_CURSORDRAG          12 //光标拖拽(触摸屏手指滑动)

#define PX\_OBJECT\_EVENT\_VALUECHANGED        13 //值改变\*

#define PX\_OBJECT\_EVENT\_DRAGFILE            14 //文件拖进

#define PX\_OBJECT\_EVENT\_KEYDOWN             15 //键盘按键按下

#define PX\_OBJECT\_EVENT\_IMPACT              16 //对象碰撞(PX\_World事件)

#define PX\_OBJECT\_EVENT\_ONFOCUSCHANGED      17 //焦点改变

#define PX\_OBJECT\_EVENT\_SCALE               18 //缩放

#define PX\_OBJECT\_EVENT\_WINDOWRESIZE        19 //窗体尺寸改变

详细内容将在之后讨论

### 获取PainterEngine绘制表面基础信息

PainterEngine的所有绘制是基于px\_surface(渲染表面)的

在PX\_ApplicationRender中pApp->runtime.RenderSurface是默认的渲染表面,在这个渲染表面绘制的图像会被直接显示在渲染结果中

因为显示设备的不一样,或者用户调整显示窗口的大小,默认渲染表面的长度和宽度可能会发生变化

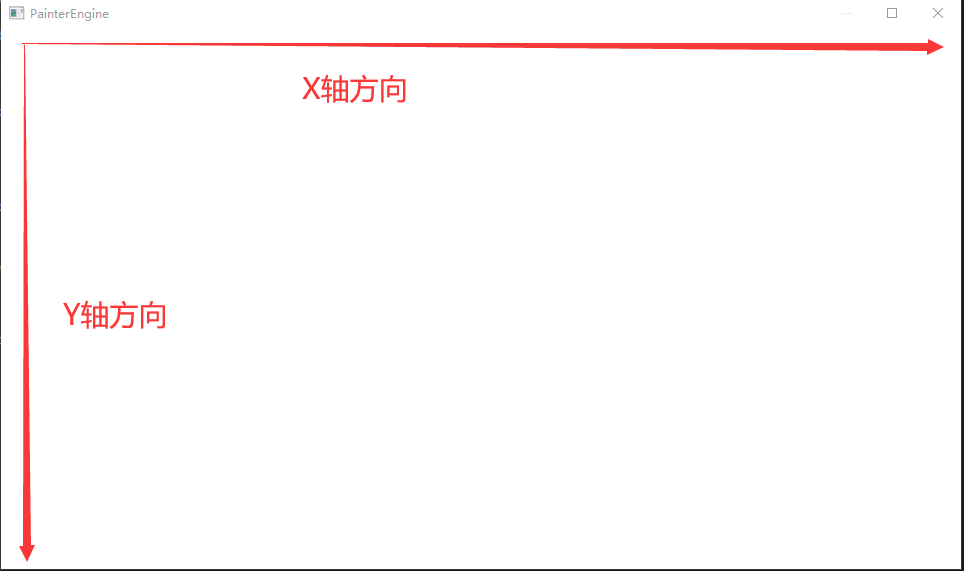
你可以使用

pApp->runtime.surface\_width 来获取当前渲染表面的像素宽度

pApp->runtime.surface\_height来获取当前渲染表面的像素高度

### 使用PainterEngine绘制几何图形

\*PainterEngine中使用的二维坐标系布置如下（屏幕坐标系）



其中，左上角为原点（0,0）右下角为终点（surface宽度，surface高度）

\*PainterEngine采用了反走样算法进行几何绘制

#### 绘制圆形

#### 线

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawLine(px\_surface \*psurface, px\_int x0, px\_int y0, px\_int x1, px\_int y1 ,pt\_int lineWidth, px\_color color); |
| 说明 | **绘制一个反走样线段** |
| 参数 | psurface 渲染表面  x0 y0起始点坐标  x1 y1 终点坐标  lineWidth 线宽  color 颜色 |

##### 示范

在PX\_ApplicationRender中添加画线代码（如下所示），分别绘制三根:

（100,100）---> (600,100) 的 宽度为2的 红线

（100,200）---> (600,200) 的 宽度为6的 绿线

（100,300）---> (600,300) 的 宽度为12的 蓝线

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

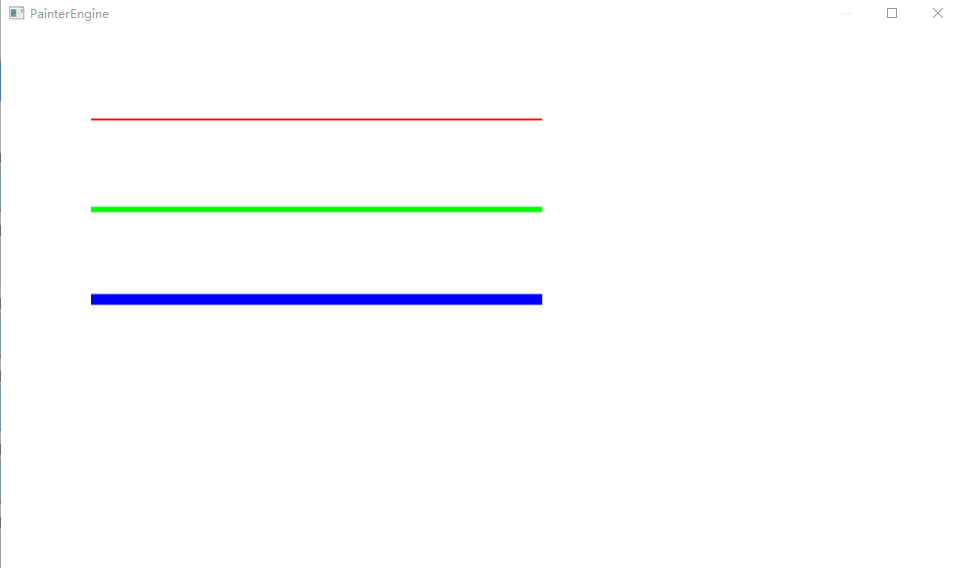
    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawLine(pRenderSurface,100,100,600,100,2,PX\_COLOR(255,255,0,0));

    PX\_GeoDrawLine(pRenderSurface,100,200,600,200,6,PX\_COLOR(255,0,255,0));

    PX\_GeoDrawLine(pRenderSurface,100,300,600,300,12,PX\_COLOR(255,0,0,255));

}



#### 边框

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawBorder(px\_surface \*psurface, px\_int left, px\_int top, px\_int rignt, px\_int bottom ,px\_int lineWidth,px\_color color); |
| 说明 | **绘制一个边框** |
| 参数 | psurface 渲染表面  left top right bottom 位置描述  lineWidth 边框宽度像素  color 颜色 |

##### 示范1

在PX\_ApplicationRender中添加代码（如下所示），分别绘制一个:

（100,100）---> (300,300) 的 宽度为3的 黑色 边框

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

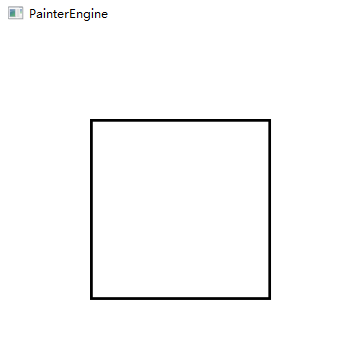
{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawBorder(pRenderSurface,100,100,300,300,3,PX\_COLOR(255,0,0,0));

}



##### 示范2

在PX\_ApplicationRender中添加代码（如下所示），分别绘制一个:

（100,100）---> (300,300) 的 宽度为8的 红色 半透明 边框

（200,200）---> (500,500) 的 宽度为8的 蓝色 半透明 边框

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

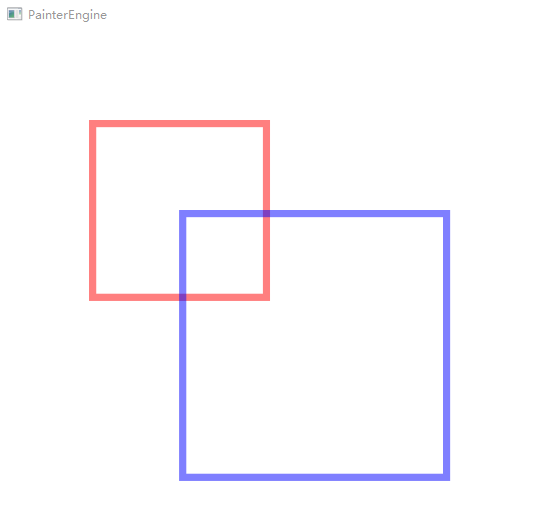
    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawBorder(pRenderSurface,100,100,300,300,8,PX\_COLOR(128,255,0,0));

    PX\_GeoDrawBorder(pRenderSurface,200,200,500,500,8,PX\_COLOR(128,0,0,255));

}



#### 矩形

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawRect(px\_surface \*psurface, px\_int left, px\_int top, px\_int right, px\_int bottom,px\_color color); |
| 说明 | **绘制一个实心矩形** |
| 参数 | psurface 渲染表面  left top right bottom 位置描述  color 颜色 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），分别绘制一个:

（100,100）---> (300,300) 的 宽度为8的 红色 半透明 矩形

（200,200）---> (500,500) 的 宽度为8的 蓝色 半透明 矩形

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

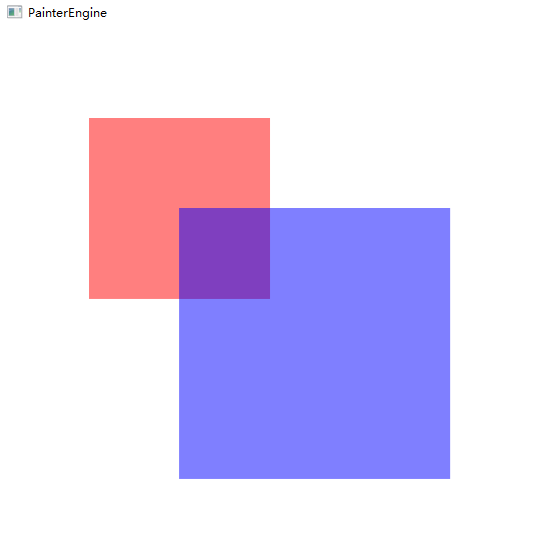
    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawRect(pRenderSurface,100,100,300,300,PX\_COLOR(128,255,0,0));

    PX\_GeoDrawRect(pRenderSurface,200,200,500,500,PX\_COLOR(128,0,0,255));

}



#### 实心圆

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawSolidCircle(px\_surface \*psurface, px\_int x,px\_int y,px\_int Radius,px\_color color ); |
| 说明 | **绘制一个反走样实心圆** |
| 参数 | psurface 渲染表面  x，y 圆心  radius 半径  color 颜色 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制一个:

(300,300) 的 半径为200的 紫色 圆形

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

     px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawSolidCircle(pRenderSurface,300,300,200,PX\_COLOR(255,255,0,255));

}

#### 圆

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawCircle(px\_surface \*psurface,px\_int x,px\_int y,px\_int Radius ,pt\_int lineWidth,px\_color color); |
| 说明 | **绘制一个反走样圆** |
| 参数 | psurface 渲染表面  x，y 圆心  radius 半径  lineWidth 线宽  color 颜色 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制一个:

(300,300) 的 半径为200 线宽为10的 紫色 圆形

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

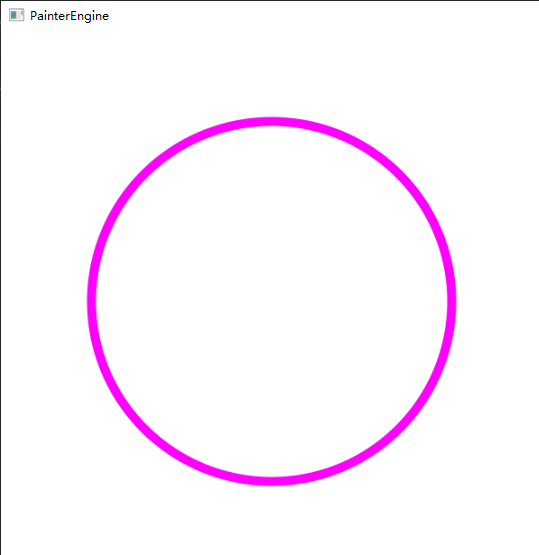
{

     px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawCircle(pRenderSurface,300,300,200,10,PX\_COLOR(255,255,0,255));

}



#### 环

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawRing(px\_surface \*psurface, px\_int x,px\_int y,px\_int Radius,px\_int lineWidth,px\_color color,px\_uint start\_angle,px\_uint end\_angle) |
| 说明 | **绘制一个反走样环** |
| 参数 | psurface 渲染表面  x，y 环心  radius 半径  lineWidth 线宽  color 颜色  start\_angle 起始角度(大于0)  end\_angle 终止角度(大于0)  这个环遵循顺时针方向 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制一个:

环心在(300,300) 的 半径为200 线宽为10 开始角度45度终止角度270度的 蓝色 环形

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

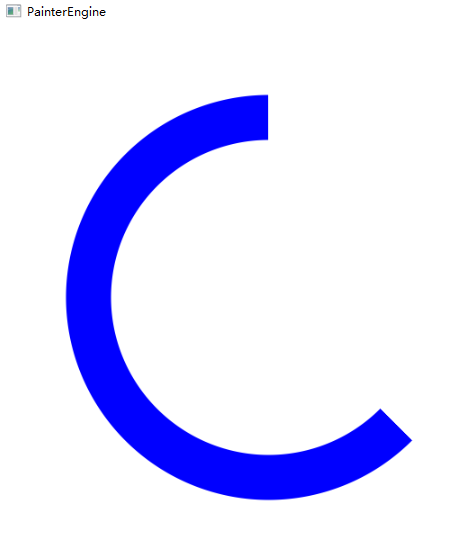
{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawRing(pRenderSurface,300,300,200,50,PX\_COLOR(255,0,0,255),45,270);

}



#### 扇形

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawSector(px\_surface \*psurface, px\_int x,px\_int y,px\_int Radius\_outside,px\_int Radius\_inside,px\_color color,px\_uint start\_angle,px\_uint end\_angle); |
| 说明 | **绘制一个反走样扇形** |
| 参数 | psurface 渲染表面  x，y 环心  outside 外径  inside 内径  color 颜色  start\_angle 起始角度  end\_angle 终止角度  这个扇形遵循顺时针方向 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制一个:

扇心在(300,300) 的 内径为0 外径为200 开始角度45度终止角度270度的 蓝色 扇形

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

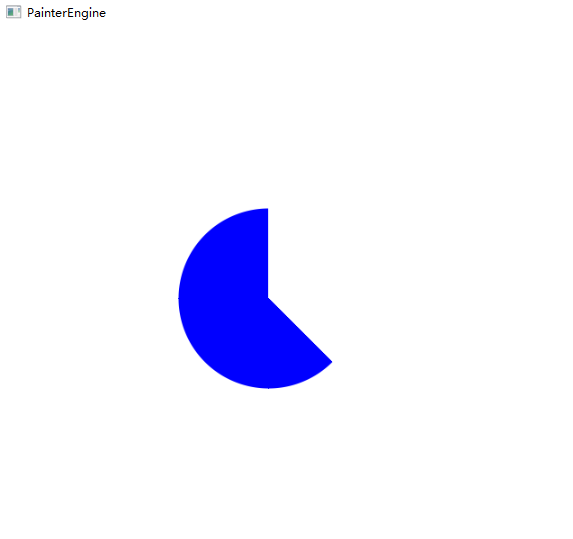
{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_GeoDrawSector(pRenderSurface,300,300,200,0,PX\_COLOR(255,0,0,255),45,270);

}



#### 圆角矩形

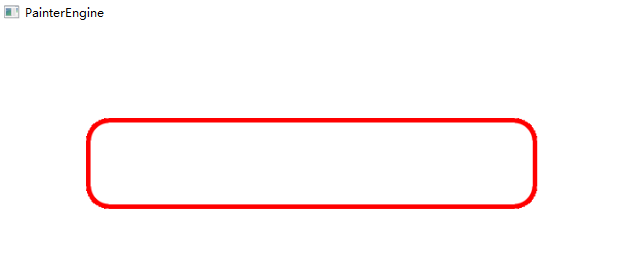
##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawRoundRect(px\_surface \*psurface, px\_int left, px\_int top, px\_int right, px\_int bottom,px\_float roundRaduis,px\_float linewidth,px\_color color); |
| 说明 | **绘制一个反走样圆角矩形** |
| 参数 | psurface 渲染表面  left top right bottom 位置描述  roundRadius 圆角半径  lineWidth 线宽  color 颜色 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制一个:

(100,100)🡪(600,200) 圆角半径为25,线宽为5的红色圆角矩形



#### 实心圆角矩形

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_GeoDrawSolidRoundRect(px\_surface \*psurface, px\_int left, px\_int top, px\_int right, px\_int bottom,px\_float roundRaduis,px\_color color); |
| 说明 | **绘制一个反走样实心圆角矩形** |
| 参数 | psurface 渲染表面  left top right bottom 位置描述  roundRadius 圆角半径  color 颜色 |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制一个:

(100,100)🡪(600,200) 圆角半径为25的蓝色实心圆角矩形



### 使用PainterEngine绘制图片纹理

#### 加载纹理

在绘制图片纹理前PainterEngine需要先将纹理加载到内存中

PainterEngine\_startup.c中,默认提供了一系列从文件加载资源的函数实现,当然这需要对应平台提供文件系统的支持.

在加载纹理之前,先定义px\_texture用于存储纹理信息,为此,打开PainterEngine\_Application.h在PX\_Application中定义一个px\_texture的实例MyTexture

typedef struct

{

    px\_texture MyTexture;

    PX\_Runtime runtime;

}PX\_Application;

(当然,这个MyTexture你也可以声明在其它地方,比如把它定义为全局变量,这里仅仅是作为一个例子)

现在,我们需要加载纹理,PainterEngine默认支持bmp和traw两种格式的文件,其中,traw格式文件可以由png文件转换,转换工具你可以在support/tools中找到(PainterEngine\_PngToTraw.exe)

然后你可以使用PX\_LoadTextureFromFile完成纹理的加载

px\_bool PX\_LoadTextureFromFile(px\_memorypool \*mp,px\_texture \*tex,const px\_char path[]);

其中:

Mp:表示将纹理加载后存储到的内存池,一般情况下,静态资源可以存储在pApp->runtime.mp\_resources中

Tex:px\_texture实例

Path:纹理文件的路径(windows以D盘下的test.traw)为例

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(PX\_LoadTextureFromFile(pApp->runtime.mp\_resources,&pApp->MyTexture,"D:/test.traw"))

    {

        //加载成功

    }

    else

    {

        //加载失败

    }

    return PX\_TRUE;

}

如果的你的程序使用Android平台,你可以将图片资源放在assets目录下,然后通过assets/图片名访问这个资源例如:

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(PX\_LoadTextureFromFile(&pApp->runtime.mp\_resources,&pApp->MyTexture,"asssets/test.traw"))

    {

        //加载成功

    }

    else

    {

        //加载失败

    }

    return PX\_TRUE;

}

现在,纹理资源加载完成了

#### 绘制纹理

PainterEngine提供多种纹理绘制函数

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_TextureRender(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend); |
| 说明 | **渲染一个纹理到表面** |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND; |
| 返回值 | - |

##### 示范

在PX\_ApplicationInitialize,PX\_ApplicationRender中添加代码（如下所示），绘制一个纹理

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(PX\_LoadTextureFromFile(&pApp->runtime.mp\_resources,&pApp->MyTexture,"D:/test.traw"))

    {

        //加载成功

    }

    else

    {

        //加载失败

    }

    return PX\_TRUE;

}

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_TextureRender(pRenderSurface,&pApp->MyTexture,pApp->runtime.surface\_width/2,pApp->runtime.window\_height/2,PX\_TEXTURERENDER\_REFPOINT\_CENTER,PX\_NULL);

}



##### 其它渲染函数

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_TextureRenderRotation(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend,px\_int Angle); |
| 说明 | **渲染一个纹理到表面,并顺时针旋转一个角度** |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND;  Angle 旋转的角度  注意该函数使用双线性插值滤波运算,对于性能要求较高的渲染函数可以使用PX\_TextureRenderEx,以较低的显示损失获得8倍左右的性能 |
| 返回值 | - |
| 函数名 | px\_void PX\_TextureRenderRotation\_vector(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend,px\_point p\_vector); |
| 说明 | 渲染一个纹理到表面,并顺时针旋转一个角度 |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND;  P\_vector 旋转向量,以x轴正方向为基准  注意该函数使用双线性插值滤波运算,对于性能要求较高的渲染函数可以使用PX\_TextureRenderEx,以较低的显示损失获得8倍左右的性能 |
| 返回值 | - |

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_TextureRenderRotation\_sincos(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend,px\_float sin,px\_float cos); |
| 说明 | 渲染一个纹理到表面,并顺时针旋转一个角度 |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND;  Sin cos 旋转矩阵对于sin cos值  注意该函数使用双线性插值滤波运算,对于性能要求较高的渲染函数可以使用PX\_TextureRenderEx,以较低的显示损失获得8倍左右的性能 |
| 返回值 | - |

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_TextureRenderEx(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend,px\_float scale,px\_float rotation); |
| 说明 | **渲染一个纹理到表面,注意,这个渲染方式的速度不如PX\_TextureRender,但提供了纹理的缩放和旋转(同样处于效率考虑,其缩放旋转均采用单点采样,其质量不如使用纹理缩放的函数,但速度快于这两函数,对于高精度旋转和缩放动画,采用该函数可以获得一个折中的效率,内存空间与动画效果)** |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND;  scale 缩放大小  rotation 旋转角度 |
| 返回值 | - |

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_TextureRenderEx\_sincos(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend,px\_float scale,px\_float sin,px\_float cos); |
| 说明 | **渲染一个纹理到表面,注意,这个渲染方式的速度不如PX\_TextureRender,但提供了纹理的缩放和旋转(以旋转矩阵sin cos形式,同样处于效率考虑,其缩放旋转均采用单点采样,其质量不如使用纹理缩放的函数,但速度快于这两函数,对于高精度旋转和缩放动画,采用该函数可以获得一个折中的效率,内存空间与动画效果)** |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND;  scale 缩放大小  sin 旋转矩阵sin值  cos 旋转矩阵 cos值 |
| 返回值 | - |

|  |  |
| --- | --- |
| 函数名 | px\_void PX\_TextureRenderEx\_vector(px\_surface \*psurface,px\_texture \*tex,px\_int x,px\_int y ,PX\_TEXTURERENDER\_REFPOINT refPoint,PX\_TEXTURERENDER\_BLEND \*blend,px\_float scale,px\_point p\_vector); |
| 说明 | **渲染一个纹理到表面,注意,这个渲染方式的速度不如PX\_TextureRender,但提供了纹理的缩放和旋转(以旋转矩阵vector形式,同样处于效率考虑,其缩放旋转均采用单点采样,其质量不如使用纹理缩放的函数,但速度快于这两函数,对于高精度旋转和缩放动画,采用该函数可以获得一个折中的效率,内存空间与动画效果)**  **默认的指向方向为** |
| 参数 | psurface 渲染到的表面  px\_texture需要渲染的纹理  x，y偏移量(该坐标以纹理左上角为参照)  refPoint 参考中心点，用于表示纹理绘制的相对位置  blend blend类型结构体，用于调整绘制纹理的alpha，hdr值  当它为PX\_NULL时，表示采用默认blend值  typedef struct  {  float hdr\_R; //HDR of Red  float hdr\_G; //HDR of Green  float hdr\_B; //HDR of Blue  float alpha; //Blend of alpha  }PX\_TEXTURERENDER\_BLEND;  scale 缩放大小  指向方向 |
| 返回值 | - |

### 使用PainterEngine绘制文本

#### 绘制默认ANSI字模

PainterEngine默认集成了ANSI文本字模,使用默认字模绘制文本十分简单

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | px\_int PX\_FontDrawText(px\_surface \*psurface,int x,int y,PX\_FONT\_ALIGN align,const px\_char \*Text,px\_color Color); |
| 说明 | **绘制ANSI文本** |
| 参数 | psurface 表面  x,y偏移坐标  align 对齐模式  PX\_FONT\_ALIGN\_LEFTTOP, 左上角对齐  PX\_FONT\_ALIGN\_MIDTOP,居中顶部对齐  PX\_FONT\_ALIGN\_RIGHTTOP,右上角对齐  PX\_FONT\_ALIGN\_LEFTMID,靠左居中对齐  PX\_FONT\_ALIGN\_CENTER,中心对齐  PX\_FONT\_ALIGN\_RIGHTMID,靠右居中对齐  PX\_FONT\_ALIGN\_LEFTBOTTOM,靠左底部对齐  PX\_FONT\_ALIGN\_MIDBOTTOM,居中底部对齐  PX\_FONT\_ALIGN\_RIGHTBOTTOM,靠右底部对齐  Text ANSI字符集文本指针  Color 颜色 |
| 返回值 | - |

##### 示范

在PX\_ApplicationRender中添加代码（如下所示），绘制深蓝色文本”Hello World 123”

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_FontDrawText(pRenderSurface,10,100,PX\_FONT\_ALIGN\_LEFTTOP,"Hello World 123",PX\_COLOR(255,0,0,128));

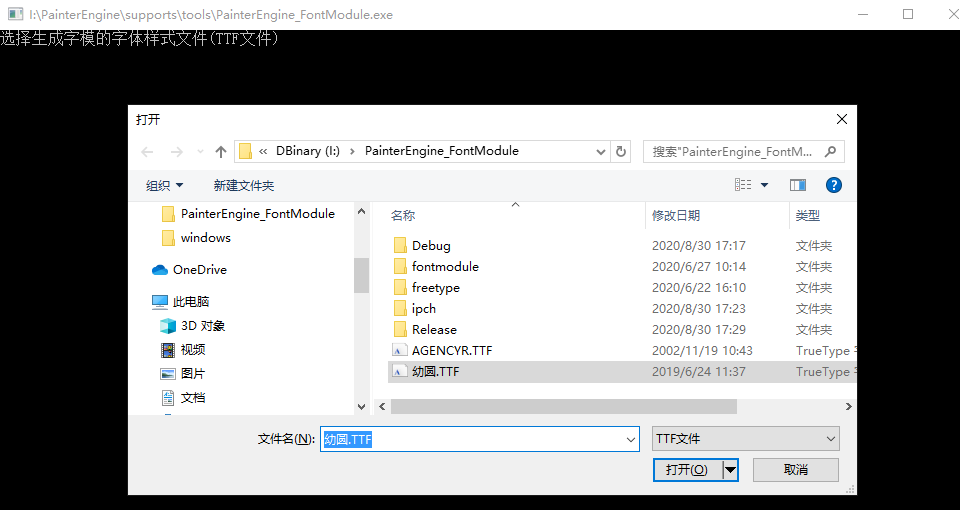
}



#### 创建字模文件

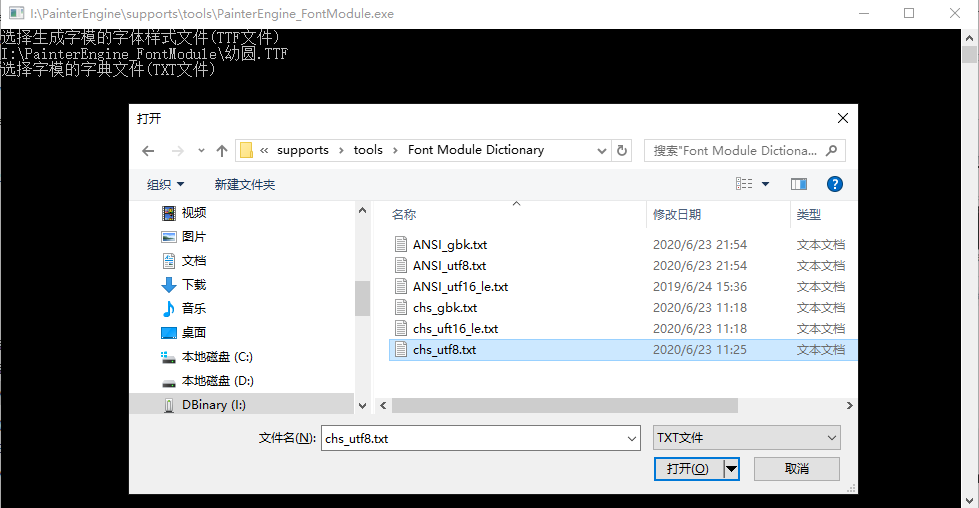
PainterEngine运行您使用TTF文件创建自定义的字模文件,字模制作工具你可以在PainterEngine/supports/tools中找到(PainterEngine\_Fontmodule.exe)

1.运行字模制作工具,选择你需要的ttf文件(以幼圆为例)

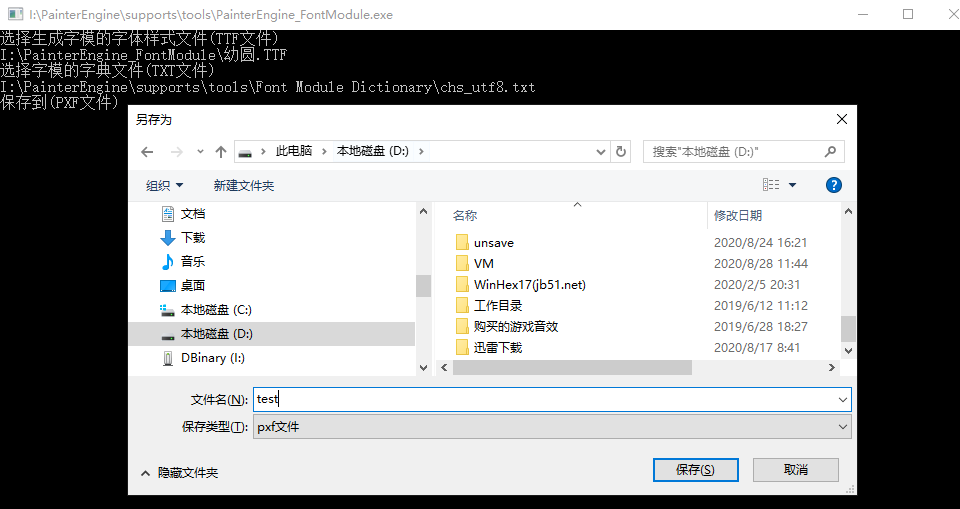


2.选择字典文件(TXT),字典文件里应该包含你程序需要用到的所有文字(如果没有,这个文字将不被显示),同时,字典文件必须按照特定的编码进行保存(强烈建议UTF8或GBK).在

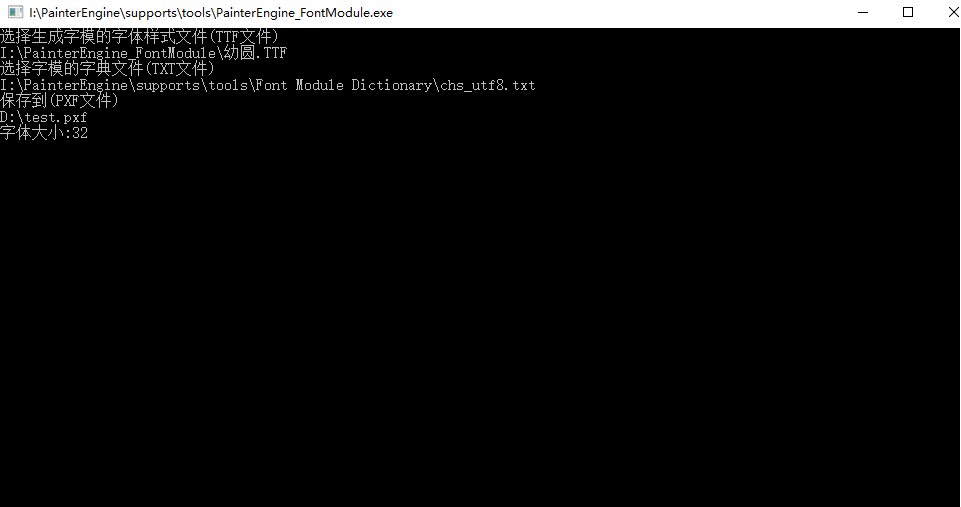
PainterEngine/supports/tools/Font Module Dictionary中,你可以找到ANSI和汉字不同编码的所有字符字典,你可以直接使用这些字典,当然,在一些存储空间紧张的情况下,强烈建议使用裁剪后的字典以减小生成字模的文件大小(这里,笔者以UTF8的字典为例).



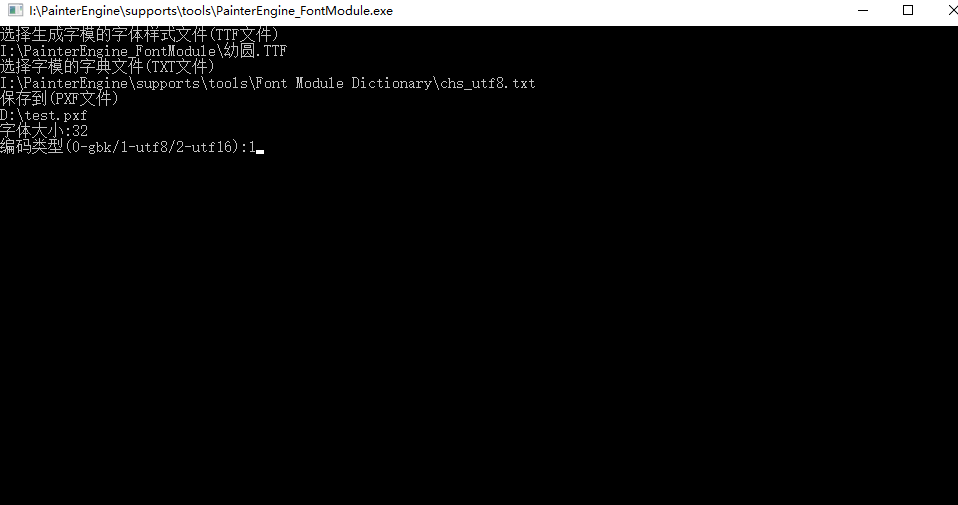
3.选择字模的生成路径(D:/test.pxf)



4.设定字体大小



5.设定编码(必须和字典文件的使用编码一致)

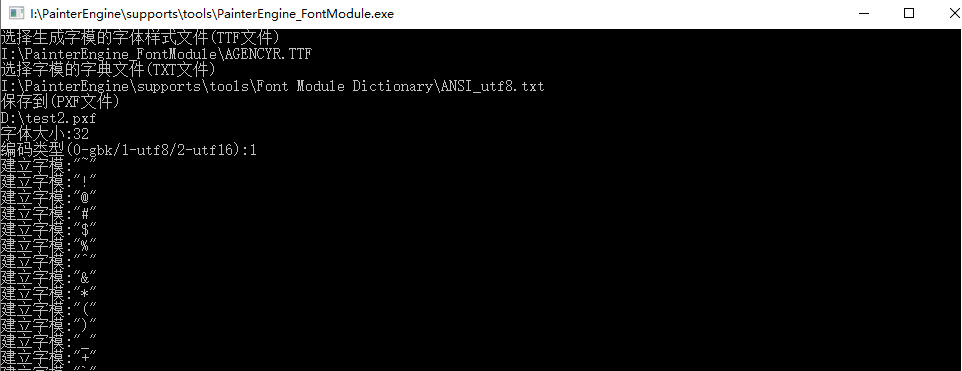


6.生成字模



现在,你可以在D盘中找到test.pxf的字模文件了

\*注意,chs中仅包含汉字文件,并不包含ANSI中的英文字符及标点,因此,如果你希望支持所有的汉字及英文标点,你应该使用ANSI\_utf8.txt和英文字体制作另一份字模文件,下面是例子:



#### 绘制字模文本

##### 相关函数原型

字模初始化

|  |  |
| --- | --- |
| 函数名 | px\_bool PX\_FontModuleInitialize(px\_memorypool \*mp,PX\_FontModule \*module,PX\_FONTMODULE\_CODEPAGE codepage) |
| 说明 | **初始化一个字模库** |
| 参数 | Mp 内存池  Module 字模库  codepage 字模代码页 |
| 返回值 | 如果成功返回PX\_TRUE,否者返回PX\_FALSE |

以字模库绘制文本

|  |  |
| --- | --- |
| 函数名 | px\_int PX\_FontModuleDrawText(px\_surface \*psurface,PX\_FontModule \*mod,int x,int y,PX\_FONT\_ALIGN align,const px\_char \*Text,px\_color Color) |
| 说明 | **以该字模库绘制文本,注意,输入的字模必须是UTF-16 littleendia编码的** |
| 参数 | Psurface 目标表面  X 原点x坐标  Y 原点y坐标    Text 文本  Color字颜色  Mod 字模库  Align 字体对齐模式  PX\_FONT\_ALIGN\_LEFTTOP, 左上角对齐  PX\_FONT\_ALIGN\_MIDTOP,居中顶部对齐  PX\_FONT\_ALIGN\_RIGHTTOP,右上角对齐  PX\_FONT\_ALIGN\_LEFTMID,靠左居中对齐  PX\_FONT\_ALIGN\_CENTER,中心对齐  PX\_FONT\_ALIGN\_RIGHTMID,靠右居中对齐  PX\_FONT\_ALIGN\_LEFTBOTTOM,靠左底部对齐 |
| 返回值 | -绘制文本的像素宽度 |

##### 1.创建字模库

使用PainterEngine绘制字模文本,首先需要创建字模库,打开PainterEngine\_Application.h,在PX\_Application的结构体定义中,添加PX\_FontModule

typedef struct

{

    PX\_FontModule fm;

    PX\_Runtime runtime;

}PX\_Application;

##### 2.初始化字模库

打开PainterEngine\_Application.c,添加初始化字模库代码

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(!PX\_FontModuleInitialize(&pApp->runtime.mp\_resources,&pApp->fm,PX\_FONTMODULE\_CODEPAGE\_UTF8)) return PX\_FALSE;

    return PX\_TRUE;

}

##### 3.加载字模文件

打开PainterEngine\_Application.c,添加初始化字模加载代码

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(!PX\_FontModuleInitialize(&pApp->runtime.mp\_resources,&pApp->fm,PX\_FONTMODULE\_CODEPAGE\_UTF8)) return PX\_FALSE;

    if(!PX\_LoadFontModuleFromFile(&pApp->fm,"D:/test.pxf")) return PX\_FALSE;//加载中文字模

    if(!PX\_LoadFontModuleFromFile(&pApp->fm,"D:/test2.pxf")) return PX\_FALSE;//加载ANSI字模

    return PX\_TRUE;

}

##### 4.绘制字模文本

打开PainterEngine\_Application.c, 在PX\_ApplicationRender中绘制字模文本:

”你好,PainterEngine”

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(!PX\_FontModuleInitialize(&pApp->runtime.mp\_resources,&pApp->fm,PX\_FONTMODULE\_CODEPAGE\_UTF8)) return PX\_FALSE;

    if(!PX\_LoadFontModuleFromFile(&pApp->fm,"D:/test.pxf")) return PX\_FALSE;//加载中文字模

    if(!PX\_LoadFontModuleFromFile(&pApp->fm,"D:/test2.pxf")) return PX\_FALSE;//加载ANSI字模

    return PX\_TRUE;

}

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

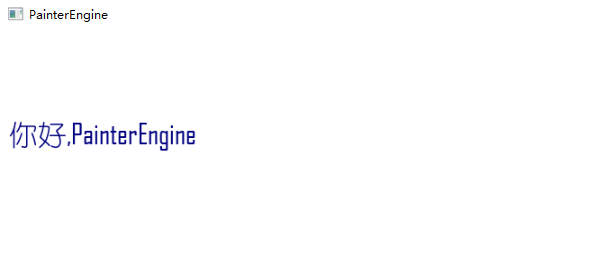
{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_FontModuleDrawText(pRenderSurface,&pApp->fm,10,100,PX\_FONT\_ALIGN\_LEFTTOP,"你好,PainterEngine",PX\_COLOR(255,0,0,128));

}



### 使用PainterEngine创建交互式组件(UI控件)

PainterEngine的应用的组建是以对象的形式存在的.PainterEngine内部集成了多种常用交互式组件(对象)供用户使用,PainterEngine强烈鼓励用户自己设计或拓展需要的功能组件(对象),下面是几个简单的交互式组件(对象)例子.

#### 创建根对象

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | **PX\_Object \*PX\_ObjectCreate(px\_memorypool \*mp,PX\_Object \*Parent,px\_float x,px\_float y,px\_float z,px\_float Width,px\_float Height,px\_float Lenght);** |
| 功能 | 创建一个对象 |
| mp | 对象使用的内存池 |
| parent | 父对象(如果为NULL表示这是一个根对象) |
| x,y,z | 坐标信息,注意z坐标信息,z的值越大,这个对象在渲染时将被最先绘制,这也就意味着z值小的对象将会覆盖在z值较大的对象中 |
| Width,Heigth,Lenght | 宽度,高度,长度 |
| 返回值 | 返回创建的对象指针 |

##### 示范

在创建其它交互式组件(对象)之前,强烈建议创建一个根对象以便于对象树的管理,创建根对象很简单,打开PainterEngine\_Application.h,定义一个PX\_Object \*ui\_root;

typedef struct

{

    PX\_Object \*ui\_root;

    PX\_FontModule fm;

    PX\_Runtime runtime;

}PX\_Application;

打开PainterEngine\_Application.c,在PX\_ApplicationInitialize中添加代码

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    if(!PX\_FontModuleInitialize(&pApp->runtime.mp\_resources,&pApp->fm,PX\_FONTMODULE\_CODEPAGE\_UTF8)) return PX\_FALSE;

    if(!PX\_LoadFontModuleFromFile(&pApp->fm,"D:/test.pxf")) return PX\_FALSE;//加载中文字模

    if(!PX\_LoadFontModuleFromFile(&pApp->fm,"D:/test2.pxf")) return PX\_FALSE;//加载ANSI字模

    pApp->ui\_root=PX\_ObjectCreate(&pApp->runtime.mp\_ui,0,0,0,0,0,0,0);

    return PX\_TRUE;

}

#### 创建按钮控件

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | PX\_Object \*PX\_Object\_PushButtonCreate(px\_memorypool \*mp,PX\_Object \*Parent,px\_int x,px\_int y,px\_int Width,px\_int Height,const px\_char \*Text,PX\_FontModule \*fontmodule,px\_color Color); |
| 说明 | **创建一个按钮交互控件** |
| 参数 | mp内存池  Parent 父对象  x,y 控件左上角位置  width 宽度  height 高度  Text 控件显示文本  fontmodule 控件显示文本的字模(为PX\_NULL表示使用ANSI默认字模)  color 文本颜色 |
| 返回值 | 如果成功PX\_Object指针,否者返回PX\_NULL |

##### 示范

打开PainterEngine\_Application.h,定义一个PX\_Object \* button;

typedef struct

{

    PX\_Object \*ui\_root;

    PX\_Object \*button;

    PX\_Runtime runtime;

}PX\_Application;

打开PainterEngine\_Application.c,在PX\_ApplicationInitialize中添加代码

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    pApp->ui\_root=PX\_ObjectCreate(&pApp->runtime.mp\_ui,0,0,0,0,0,0,0);

    pApp->button=PX\_Object\_PushButtonCreate(&pApp->runtime.mp\_ui,pApp->ui\_root,100,100,84,32,"Button",PX\_NULL,PX\_COLOR(255,0,0,0));

    return PX\_TRUE;

}

在PX\_ApplicationUpdate中添加代码

px\_void PX\_ApplicationUpdate(PX\_Application \*pApp,px\_dword elpased)

{

    PX\_ObjectUpdate(pApp->ui\_root,elpased);

}

在PX\_ApplicationRender中添加代码

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_ObjectRender(pRenderSurface,pApp->ui\_root,elpased);

}

在PX\_ApplicationPostEvent中添加代码

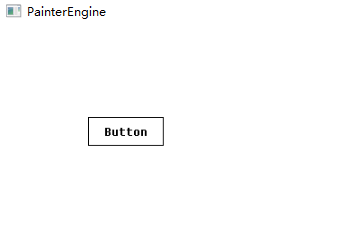
px\_void PX\_ApplicationPostEvent(PX\_Application \*pApp,PX\_Object\_Event e)

{

    PX\_ApplicationEventDefault(&pApp->runtime, e);

    PX\_ObjectPostEvent(pApp->ui\_root,e);

}



##### 添加点击事件

###### 函数原型

##### 注册对象事件

|  |  |
| --- | --- |
| 函数名 | px\_int PX\_ObjectRegisterEvent(PX\_Object \*Object,px\_uint Event,px\_void (\*ProcessFunc)(PX\_Object \*,PX\_Object\_Event e,px\_void \*user\_ptr),px\_void \*ptr); |
| 功能 | 为一个对象注册响应事件 |
| 参数Object | 需要绘制的对象指针 |
| 参数Event | 响应的事件类型 |
| ProcessFunc | 响应处理函数 |
| ptr | 用户自定义指针 |

###### 示范

首先编写按钮点击处理函数, 打开PainterEngine\_Application.c添加下面的代码

#include "windows.h"

px\_void OnButtonExecute(PX\_Object \*pObject,PX\_Object\_Event e,px\_void \*ptr)

{

    MessageBox(0,"Button Clicked","",MB\_OK);

}

然后将处理函数和对象事件进行绑定

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

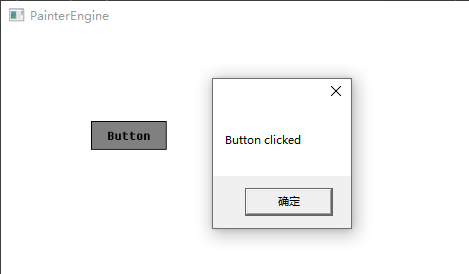
    pApp->ui\_root=PX\_ObjectCreate(&pApp->runtime.mp\_ui,0,0,0,0,0,0,0);

    pApp->button=PX\_Object\_PushButtonCreate(&pApp->runtime.mp\_ui,pApp->ui\_root,100,100,84,32,"Button",PX\_NULL,PX\_COLOR(255,0,0,0));

    PX\_ObjectRegisterEvent(pApp->button,PX\_OBJECT\_EVENT\_EXECUTE,OnButtonExecute,pApp);//注册按钮事件

    return PX\_TRUE;

}



###### 完整代码

#include "PainterEngine\_Application.h"

PX\_Application App;

#include "windows.h"

px\_void OnButtonExecute(PX\_Object \*pObject,PX\_Object\_Event e,px\_void \*ptr)

{

    MessageBox(0,"Button clicked","",MB\_OK);

}

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    pApp->ui\_root=PX\_ObjectCreate(&pApp->runtime.mp\_ui,0,0,0,0,0,0,0);//创建根对象

    pApp->button=PX\_Object\_PushButtonCreate(&pApp->runtime.mp\_ui,pApp->ui\_root,100,100,84,32,"Button",PX\_NULL,PX\_COLOR(255,0,0,0));

    PX\_ObjectRegisterEvent(pApp->button,PX\_OBJECT\_EVENT\_EXECUTE,OnButtonExecute,pApp);//注册按钮事件

    return PX\_TRUE;

}

px\_void PX\_ApplicationUpdate(PX\_Application \*pApp,px\_dword elpased)

{

    PX\_ObjectUpdate(pApp->ui\_root,elpased);

}

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_ObjectRender(pRenderSurface,pApp->ui\_root,elpased);

}

px\_void PX\_ApplicationPostEvent(PX\_Application \*pApp,PX\_Object\_Event e)

{

    PX\_ApplicationEventDefault(&pApp->runtime, e);

    PX\_ObjectPostEvent(pApp->ui\_root,e);

}

#### 创建文本控件

##### 函数原型

|  |  |
| --- | --- |
| 函数名 | PX\_Object\* PX\_Object\_EditCreate(px\_memorypool \*mp, PX\_Object \*Parent,px\_int x,px\_int y,px\_int Width,px\_int Height,PX\_FontModule \*fontModule,px\_color TextColor ) |
| 说明 | **创建一个编辑框控件** |
| 参数 | mp内存池  Parent 父对象  x,y 控件左上角位置  width 宽度  height 高度  fontmodule 控件显示文本的字模(为PX\_NULL表示使用ANSI默认字模)  color 文本颜色 |
| 返回值 | 如果成功PX\_Object指针,否者返回PX\_NULL |

##### 示范

打开PainterEngine\_Application.h,定义一个PX\_Object \* Edit;

typedef struct

{

    PX\_Object \*ui\_root;

    PX\_Object \* Edit;

    PX\_Runtime runtime;

}PX\_Application;

打开PainterEngine\_Application.c,在PX\_ApplicationInitialize中添加代码

px\_bool PX\_ApplicationInitialize(PX\_Application \*pApp,px\_int screen\_width,px\_int screen\_height)

{

    PX\_ApplicationInitializeDefault(&pApp->runtime, screen\_width, screen\_height);

    pApp->ui\_root=PX\_ObjectCreate(&pApp->runtime.mp\_ui,0,0,0,0,0,0,0);//创建根对象

    pApp->edit=PX\_Object\_EditCreate(&pApp->runtime.mp\_ui,pApp->ui\_root,100,100,128,32,PX\_NULL,PX\_COLOR(255,0,0,0));

    return PX\_TRUE;

}

在PX\_ApplicationUpdate中添加代码

px\_void PX\_ApplicationUpdate(PX\_Application \*pApp,px\_dword elpased)

{

    PX\_ObjectUpdate(pApp->ui\_root,elpased);

}

在PX\_ApplicationRender中添加代码

px\_void PX\_ApplicationRender(PX\_Application \*pApp,px\_dword elpased)

{

    px\_surface \*pRenderSurface=&pApp->runtime.RenderSurface;

    PX\_RuntimeRenderClear(&pApp->runtime,PX\_COLOR(255,255,255,255));

    PX\_ObjectRender(pRenderSurface,pApp->ui\_root,elpased);

}

在PX\_ApplicationPostEvent中添加代码

px\_void PX\_ApplicationPostEvent(PX\_Application \*pApp,PX\_Object\_Event e)

{

    PX\_ApplicationEventDefault(&pApp->runtime, e);

    PX\_ObjectPostEvent(pApp->ui\_root,e);

}

