基于c++，OpenGL渲染的碰撞游戏

一：准备工作：

1. 安装opengl渲染库，SQIL库，并且进行相关文件路径设置工作，具体过程就不详细叙述了。
2. 写好基本框架：

   首先定义一个游戏类，它会包含所有相关的渲染和游戏代码

   游戏类封装了一个初始化函数，一个更新函数，一个处理输入函数以及一个渲染函数

```
class Game
{
    public:
        // 游戏状态
        GameState  State;
        GLboolean  Keys[1024];
        GLuint     Width, Height;
        // 构造函数/析构函数
        Game(GLuint width, GLuint height);
        ~Game();
        // 初始化游戏状态（加载所有的着色器/纹理/关卡）
        void Init();
        // 游戏循环
        void ProcessInput(GLfloat dt);
        void Update(GLfloat dt);
        void Render();
};
```

这个类应该包含了所有在一个游戏类中会出现的东西。我们通过给定一个宽度和高度来初始化这个游戏，并且使用Init函数来加载着色器，纹理并且初始化所有的游戏状态。我们可以通过调用ProcessInput函数，并且使用储存在Keys数组里的数据来处理输入。并且在Update里面更新游戏状态（玩家与球的运动）。最后，我们还可以调用Render函数来对游戏进行渲染。渲染与游戏是分开的。

在game类里面同样封装了一个叫做state的变量，类型为Gamestate,定义如下：

```
// 代表了游戏的当前状态
enum GameState {
    GAME_ACTIVE,
    GAME_MENU,
    GAME_WIN
};
```

这个类可以帮助我们跟踪游戏当前状态。

这样的话我们就可以根据当前游戏的状态来决定渲染和/或者处理不同的元素(Item)了（比如当我们在游戏菜单界面的时候就可能需要渲染和处理不同的元素了）。

目前为止，这个游戏类的函数还完全是空的，因为我们还没有写游戏的实际代码。

3.工具类

由于在开发过程中，我们要频繁使用一些OPENGL的概念，比如纹理和着色器。因此为了这两个元素创建一个更加易用的接口十分必要。着色器类会接受两个或者三个字符串，并且生成一个编译好的着色器。具体原理比较繁杂就不阐述了。

着色器，纹理代码在包里，shader.c;texture.c

4.资源管理

尽管着色器与纹理类的函数本身就很棒了，它们仍需要有一个字节数组或一些字符串来调用它们。我们可以很容易将文件加载代码嵌入到它们自己的类中，但这稍微有点违反了单一功能原则(Single Responsibility Principle)，即这两个类应当分别仅仅关注纹理或者着色器本身，而不是它们的文件加载机制。

文件：resource_manager.c

5.程序

我们仍然需要为这个游戏创建一个窗口并且设置一些OpenGL的初始状态

这个Breakout游戏的起始代码非常简单：我们用GLFW创建一个窗口，注册一些回调函数，创建一个Game对象，并将所有相关的信息都传到游戏类中。代码如下：

代码见：game.c

```
```

二：渲染精灵

这个部分主要对图片以及一些效果的渲染，参照opengl教程，对于渲染精灵这里不作阐述，可以参照opengl的开发文档。

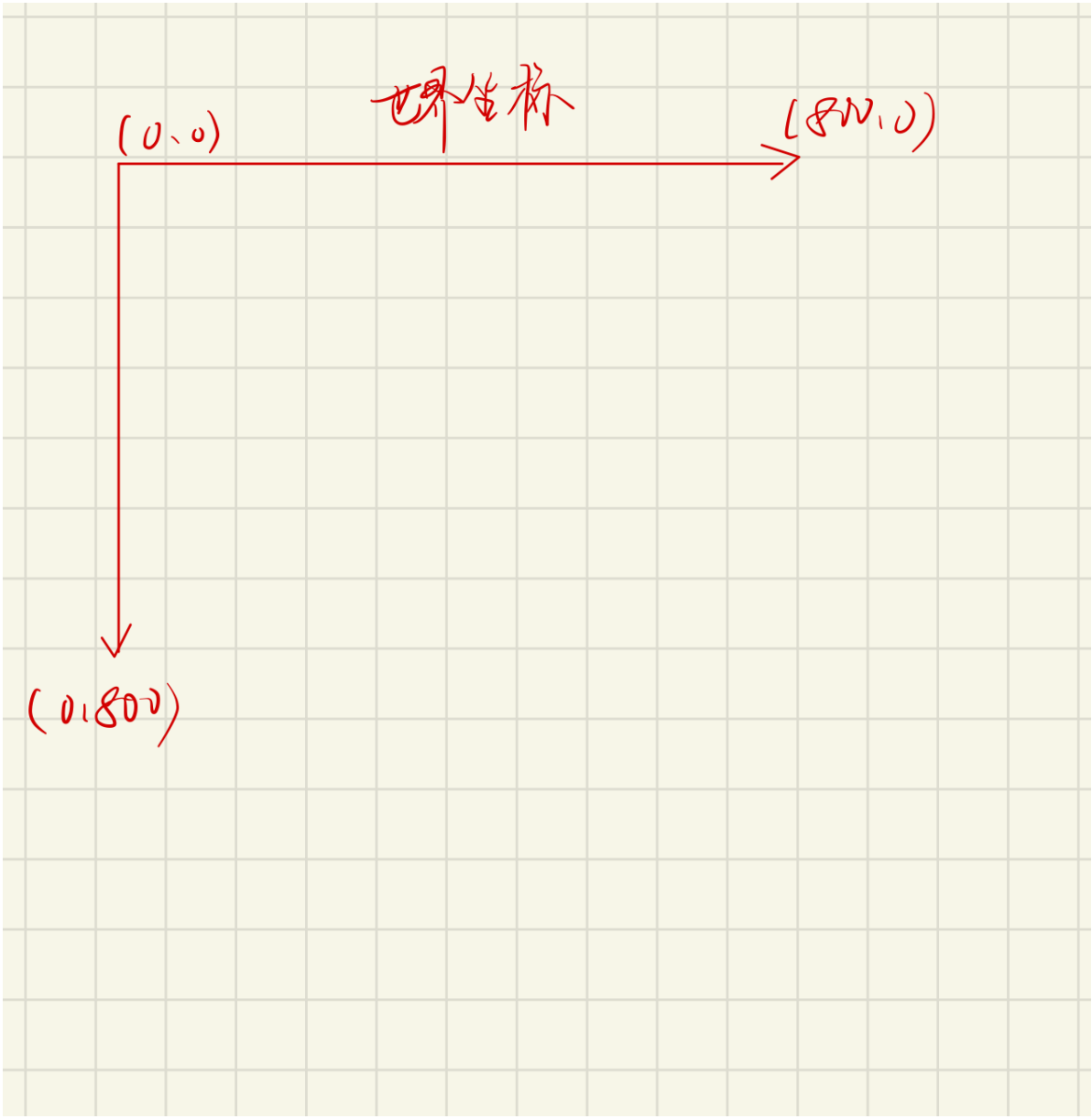
流程大致为:

2D投影矩阵--->渲染精灵------>初始化------->渲染

由于游戏是2d不需要透视：我们可以定义如下的投影矩阵指定世界坐标为屏幕坐标

1.2D投影矩阵

```
glm::mat4 projection = glm::ortho(0.0f, 800.0f, 600.0f, 0.0f, -1.0f, 1.0f);
```



2.渲染精灵

渲染一个实际的精灵应该不会太复杂。我们创建一个有纹理的四边形，它在之后可以使用一个模型矩阵来变换，然后我们会用之前定义的正射投影矩阵来投影它。

```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 position, vec2 texCoords>

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 projection;

void main()
{
    TexCoords = vertex.zw;
    gl_Position = projection * model * vec4(vertex.xy, 0.0, 1.0);
}
```

为了让精灵的渲染更加有条理，我们定义了一个SpriteRenderer类，有了它只需要一个函数就可以渲染精灵了。它的定义如下：

```
class SpriteRenderer
{
    public:
        SpriteRenderer(Shader &shader);
        ~SpriteRenderer();

        void DrawSprite(Texture2D &texture, glm::vec2 position,
            glm::vec2 size = glm::vec2(10, 10), GLfloat rotate = 0.0f,
            glm::vec3 color = glm::vec3(1.0f));
    private:
        Shader shader;
        GLuint quadVAO;

        void initRenderData();
};
```

3.初始化

我们首先定义了一组以四边形的左上角为(0,0)坐标的顶点。这意味着当我们在四边形上应用一个位移或缩放变换的时候，它们会从四边形的左上角开始进行变换

```
void SpriteRenderer::initRenderData()
{
    // 配置 VAO/VBO
    GLuint VBO;
    GLfloat vertices[] = {
        // 位置     // 纹理
        0.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f,

        0.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, 0.0f, 1.0f, 0.0f
    };

    glGenVertexArrays(1, &this->quadVAO);
    glGenBuffers(1, &VBO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glBindVertexArray(this->quadVAO);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat),
(GLvoid*)0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

4.渲染

渲染精灵并不是太难；我们使用精灵渲染器的着色器，配置一个模型矩阵并且设置相关的uniform。这里最重要的就是变换的顺序：当试图在一个场景中用旋转矩阵和缩放矩阵放置一个对象的时候，建议是首先做缩放变换，再旋转，最后才是位移变换。因为矩阵乘法是从右向左执行的，所以我们变换的矩阵顺序是相反的：移动，旋转，缩放。

可能写的有点蒙逼，但实际渲染就是为了加载图片。具体原理可自己查阅文档。

三：关卡

我们希望这些关卡有以下特性：他们足够灵活以便于支持任意数量的行或列、可以拥有不可摧毁的坚固砖块、支持多种类型的砖块且这些信息被存储在外部文件中。

在本教程中，我们将简要介绍用于管理大量砖块的游戏关卡对象的代码，首先我们需要先定义什么是一个砖块。

关卡基本由砖块组成，因此我们可以用一个砖块的集合表示一个关卡。因为砖块需要和游戏对象几乎相同的状态，所以我们将关卡中的每个砖块表示为GameObject。GameLevel类的布局如下所示：

```
class GameLevel
{
public:
    std::vector<GameObject> Bricks;

    GameLevel() { }
    // 从文件中加载关卡
    void Load(const GLchar *file, GLuint levelWidth, GLuint levelHeight);
    // 渲染关卡
    void Draw(SpriteRenderer &renderer);
    // 检查一个关卡是否已完成 (所有非坚硬的瓷砖均被摧毁)
    GLboolean IsCompleted();
private:
    // 由砖块数据初始化关卡
    void init(std::vector<std::vector<GLuint>> tileData, GLuint levelWidth,
GLuint levelHeight);
};
```

由于关卡数据从外部文本中加载，所以我们需要提出某种关卡的数据结构，以下是关卡数据在文本文件中可能的表示形式的一个例子：

```
1 1 1 1 1 1
2 2 0 0 2 2
3 3 4 4 3 3
```

在这里一个关卡被存储在一个矩阵结构中，每个数字代表一种类型的砖块，并以空格分隔。在关卡代码中我们可以假定每个数字代表什么：

- 数字0：无砖块，表示关卡中空的区域
- 数字1：一个坚硬的砖块，不可被摧毁
- 大于1的数字：一个可被摧毁的砖块，不同的数字区分砖块的颜色

上面的示例关卡在被GameLevel处理后，看起来会像这样：

GameLevel类使用两个函数从文件中生成一个关卡。它首先将所有数字在Load函数中加载到二维容器(vector)里，然后在init函数中处理这些数字，以创建所有的游戏对象。

```
void GameLevel::Load(const GLchar *file, GLuint levelWidth, GLuint levelHeight)
{
    // 清空过期数据
    this->Bricks.clear();
    // 从文件中加载
    GLuint tileCode;
    GameLevel level;
    std::string line;
    std::ifstream fstream(file);
    std::vector<std::vector<GLuint>> tileData;
    if (fstream)
    {
        while (std::getline(fstream, line)) // 读取关卡文件的每一行
        {
            std::istringstream sstream(line);
            std::vector<GLuint> row;
            while (sstream >> tileCode) // 读取被空格分隔的每个数字
                row.push_back(tileCode);
            tileData.push_back(row);
        }
        if (tileData.size() > 0)
            this->init(tileData, levelWidth, levelHeight);
    }
}
```

被加载后的tileData数据被传递到GameLevel的init函数：

```
void GameLevel::init(std::vector<std::vector<GLuint>> tileData, GLuint lvlWidth,
GLuint lvlHeight)
{
    // 计算每个维度的大小
    GLuint height = tileData.size();
    GLuint width = tileData[0].size();
    GLfloat unit_width = lvlWidth / static_cast<GLfloat>(width);
    GLfloat unit_height = lvlHeight / height;
    // 基于tileDataC初始化关卡
    for (GLuint y = 0; y < height; ++y)
    {
        for (GLuint x = 0; x < width; ++x)
        {
            // 检查砖块类型
            if (tileData[y][x] == 1)
            {
                glm::vec2 pos(unit_width * x, unit_height * y);
                glm::vec2 size(unit_width, unit_height);
                GameObject obj(pos, size,
                    ResourceManager::GetTexture("block_solid"),
                    glm::vec3(0.8f, 0.8f, 0.7f)
                );
                obj.IsSolid = GL_TRUE;
                this->Bricks.push_back(obj);
            }
            else if (tileData[y][x] > 1)
```

```
            {
                glm::vec3 color = glm::vec3(1.0f); // 默认为白色
                if (tileData[y][x] == 2)
                    color = glm::vec3(0.2f, 0.6f, 1.0f);
                else if (tileData[y][x] == 3)
                    color = glm::vec3(0.0f, 0.7f, 0.0f);
                else if (tileData[y][x] == 4)
                    color = glm::vec3(0.8f, 0.8f, 0.4f);
                else if (tileData[y][x] == 5)
                    color = glm::vec3(1.0f, 0.5f, 0.0f);

                glm::vec2 pos(unit_width * x, unit_height * y);
                glm::vec2 size(unit_width, unit_height);
                this->Bricks.push_back(
                    GameObject(pos, size, ResourceManager::GetTexture("block"),
color)
                );
            }
        }
    }
}
```

init函数遍历每个被加载的数字，处理后将一个相应的GameObject添加到关卡的容器中。每个砖块的尺寸(unit_width和unit_height)根据砖块的总数被自动计算以便于每块砖可以完美地适合屏幕边界。

在这里我们用两个新的纹理加载游戏对象，分别为[block]纹理与[solid block]纹理

这里有一个很好的小窍门，即这些纹理是完全灰度的。其效果是，我们可以在游戏代码中，通过将灰度值与定义好的颜色矢量相乘来巧妙地操纵它们的颜色，就如同我们在SpriteRenderer中所做的那样。这样一来，自定义的颜色/外观就不会显得怪异或不平衡。

游戏中

我们希望在Breakout游戏中支持多个关卡，因此我们将在Game类中添加一个持有GameLevel变量的容器。同时我们还将存储当前的游戏关卡

```
class Game
{
    [...]
    std::vector<GameLevel> Levels;
    GLuint                 Level;
    [...]
};
```

这个教程的Breakout版本共有4个游戏关卡：

- [Standard]
- [A few small gaps]
- [Space invader]
- [Bounce galore]

然后Game类的init函数初始化每个纹理和关卡：

```
void Game::Init()
{
```

```
    [...]
    // 加载纹理
    ResourceManager::LoadTexture("textures/background.jpg", GL_FALSE,
"background");
    ResourceManager::LoadTexture("textures/awesomeface.png", GL_TRUE, "face");
    ResourceManager::LoadTexture("textures/block.png", GL_FALSE, "block");
    ResourceManager::LoadTexture("textures/block_solid.png", GL_FALSE,
"block_solid");
    // 加载关卡
    GameLevel one; one.Load("levels/one.lvl", this->Width, this->Height * 0.5);
    GameLevel two; two.Load("levels/two.lvl", this->Width, this->Height * 0.5);
    GameLevel three; three.Load("levels/three.lvl", this->Width, this->Height *
0.5);
    GameLevel four; four.Load("levels/four.lvl", this->Width, this->Height *
0.5);
    this->Levels.push_back(one);
    this->Levels.push_back(two);
    this->Levels.push_back(three);
    this->Levels.push_back(four);
    this->Level = 1;
}
```

现在剩下要做的就是通过调用当前关卡的Draw函数来渲染我们完成的关卡，然后使用给定的sprite渲染器调用每个GameObject的Draw函数。除了关卡之外，我们还会用一个很好的[背景图片]来渲染这个场景：

```
void Game::Render()
{
    if(this->State == GAME_ACTIVE)
    {
        // 绘制背景
        Renderer->DrawSprite(ResourceManager::GetTexture("background"),
            glm::vec2(0, 0), glm::vec2(this->Width, this->Height), 0.0f
        );
        // 绘制关卡
        this->Levels[this->Level].Draw(*Renderer);
    }
}
```

玩家挡板

此时我们在场景底部引入一个由玩家控制的挡板，挡板只允许水平移动，并且在它接触任意场景边缘时停止。对于玩家挡板，我们将使用以下纹理：

一个挡板对象拥有位置、大小、渲染纹理等属性，所以我们理所当然地将其定义为一个GameObject。

```
// 初始化挡板的大小
const glm::vec2 PLAYER_SIZE(100, 20);
// 初始化当班的速率
const GLfloat PLAYER_VELOCITY(500.0f);

GameObject      *Player;
```

```
void Game::Init()
{
    [...]
    ResourceManager::LoadTexture("textures/paddle.png", true, "paddle");
    [...]
    glm::vec2 playerPos = glm::vec2(
        this->Width / 2 - PLAYER_SIZE.x / 2,
        this->Height - PLAYER_SIZE.y
    );
    Player = new GameObject(playerPos, PLAYER_SIZE,
ResourceManager::GetTexture("paddle"));
}
```

这里我们定义了几个常量来初始化挡板的大小与速率。在Game的Init函数中我们计算挡板的初始位置，使其中心与场景的水平中心对齐。

除此之外我们还需要在Game的Render函数中添加：

```
Player->Draw(*Renderer);
```

如果你现在启动游戏，你不仅会看到关卡画面，还会有一个在场景底部边缘的奇特的挡板。到目前为止，它除了静态地放置在那以外不会发生任何事情，因此我们需要进入游戏的ProcessInput函数，使得当玩家按下A和D时，挡板可以水平移动

```
oid Game::ProcessInput(GLfloat dt)
{
    if (this->State == GAME_ACTIVE)
    {
        GLfloat velocity = PLAYER_VELOCITY * dt;
        // 移动挡板
        if (this->Keys[GLFW_KEY_A])
        {
            if (Player->Position.x >= 0)
                Player->Position.x -= velocity;
        }
        if (this->Keys[GLFW_KEY_D])
        {
            if (Player->Position.x <= this->Width - Player->Size.x)
                Player->Position.x += velocity;
        }
    }
}
```

在这里，我们根据用户按下的键，向左或向右移动挡板(注意我们将速率与deltaTime相乘)。当挡板的x值小于0，它将移动出游戏场景的最左侧，所以我们只允许挡板的x值大于0时向左移动。对于右侧边缘我们做相同的处理，但我们必须比较场景的右侧边缘与挡板的右侧边缘，即场景宽度减去挡板宽度。

现在启动游戏，将呈现一个玩家可控制在整个场景底部自由移动的挡板

具体代码在game.c,game.h

四：球

此时我们已经有了一个包含有很多砖块和玩家的一个挡板的关卡。与经典的Breakout内容相比还差一个球。游戏的目的是让球撞击所有的砖块，直到所有的可销毁砖块都被销毁，但同时也要满足条件：球不能碰触屏幕的下边缘。

除了通用的游戏对象组件，球还需要有半径和一个布尔值，该布尔值用于指示球被固定(stuck)在玩家挡板上还是被允许自由运动的状态。当游戏开始时，球被初始固定在玩家挡板上，直到玩家按下任意键开始游戏

由于球只是一个附带了一些额外属性的GameObject，所以按照常规需要创建BallObject类作为GameObject的子类。

```
class BallObject : public GameObject
{
    public:
        // 球的状态
        GLfloat   Radius;
        GLboolean Stuck;


        BallObject();
        BallObject(glm::vec2 pos, GLfloat radius, glm::vec2 velocity, Texture2D
sprite);

        glm::vec2 Move(GLfloat dt, GLuint window_width);
        void      Reset(glm::vec2 position, glm::vec2 velocity);
};
```

BallObject的构造函数不但初始化了其自身的值，而且实际上也潜在地初始化了GameObject。BallObject类拥有一个Move函数，该函数用于根据球的速度来移动球，并检查它是否碰到了场景的任何边界，如果碰到的话就会反转球的速度：

```
glm::vec2 BallObject::Move(GLfloat dt, GLuint window_width)
{
    // 如果没有被固定在挡板上
    if (!this->Stuck)
    {
        // 移动球
        this->Position += this->Velocity * dt;
        // 检查是否在窗口边界以外，如果是的话反转速度并恢复到正确的位置
        if (this->Position.x <= 0.0f)
        {
            this->Velocity.x = -this->Velocity.x;
            this->Position.x = 0.0f;
        }
        else if (this->Position.x + this->Size.x >= window_width)
        {
            this->Velocity.x = -this->Velocity.x;
            this->Position.x = window_width - this->Size.x;
        }
        if (this->Position.y <= 0.0f)
        {
            this->Velocity.y = -this->Velocity.y;
            this->Position.y = 0.0f;
        }

    }
    return this->Position;
}
```

除了反转球的速度之外，我们还需要把球沿着边界重新放置回来。只有在没有被固定时球才能够移动。

代码见：BallObject.c

首先我们在游戏中添加球。与玩家挡板相似，我们创建一个球对象并且定义两个用来初始化球的常量。对于球的纹理，我们会使用在LearnOpenGL Breakout游戏中完美适用的一张图片：

然后我们在每帧中调用游戏代码中Update函数里的Move函数来更新球的位置：

除此之外，由于球初始是固定在挡板上的，我们必须让玩家能够从固定的位置重新移动它。我们选择使用空格键来从挡板释放球。这意味着我们必须稍微修改ProcessInput函数：

现在如果玩家按下了空格键，球的Stuck值会设置为false。我们还需要更新ProcessInput函数，当球被固定的时候，会跟随挡板的位置来移动球。

最后我们需要渲染球，此时这应该很显而易见了：

```
void Game::Render()
{
    if (this->State == GAME_ACTIVE)
    {
        [...]
        Ball->Draw(*Renderer);
    }
}
```

到这里已经实现球的运动了，下一步便是碰撞检测了。

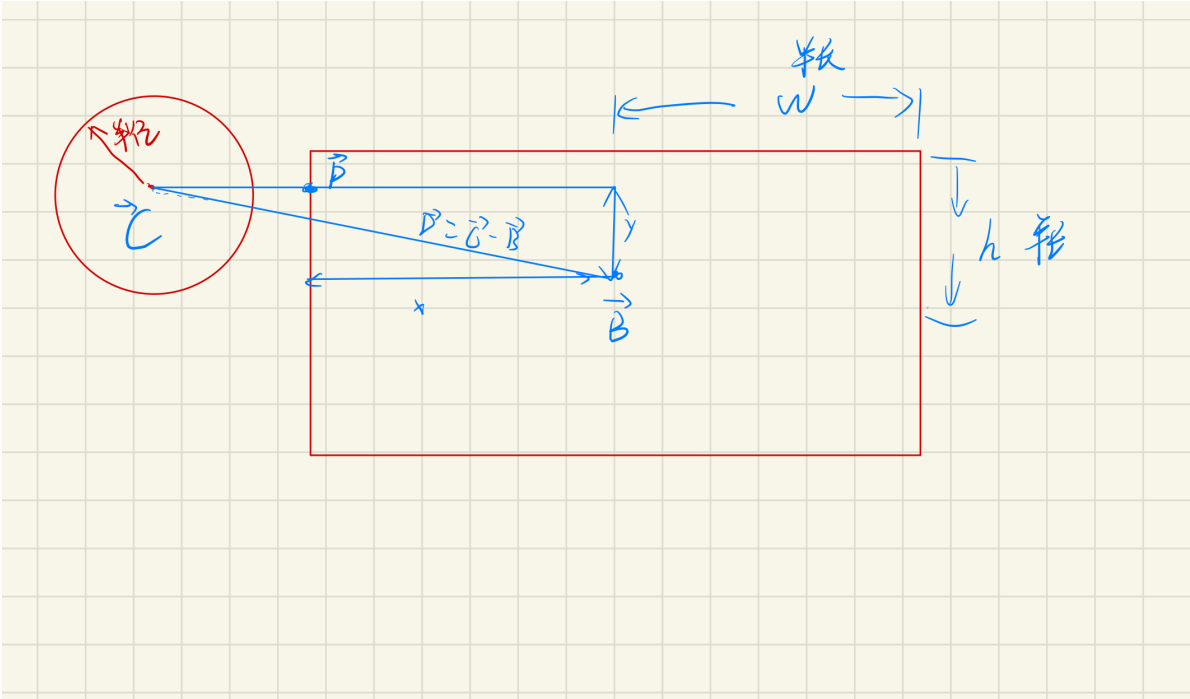
五：碰撞检测

这里就不废话了，talk is cheap,show you the code!能看懂就看，看不懂老夫也无能为力！简单解释就是判断两个物体圆（弹珠），长方体砖块，位置。

因此我们在球对象中包含了Radius变量，为了定义圆形碰撞外形，我们需要的是一个位置矢量和一个半径。

检测圆和AABB碰撞的算法会稍稍复杂，关键点如下：我们会找到AABB上距离圆最近的一个点，如果圆到这一点的距离小于它的半径，那么就产生了碰撞。

难点在于获取AABB上的最近点$\bar{P}\bar{P}$。下图展示了对于任意的AABB和圆我们如何计算该点：（AABB表示长方形）



首先我们要获取球心C¯C¯与AABB中心B¯B¯的矢量差D¯D¯。接下来用AABB的半边长(half-extents)ww和h¯h¯来限制(clamp)矢量D¯D¯。长方形的半边长是指长方形的中心到它的边的距离；简单的说就是它的尺寸除以2。这一过程返回的是一个总是位于AABB的边上的位置矢量（除非圆心在AABB内部）。

这个限制后矢量P¯P¯就是AABB上距离圆最近的点。接下来我们需要做的就是计算一个新的差矢量D'¯D'¯，它是圆心C¯C¯和P¯P¯的差矢量

既然我们已经有了矢量D'¯D'¯，我们就可以比较它的长度和圆的半径以判断是否发生了碰撞。

```
GLboolean CheckCollision(BallObject &one, GameObject &two) // AABB - Circle
collision
{
    // 获取圆的中心
    glm::vec2 center(one.Position + one.Radius);
    // 计算AABB的信息（中心、半边长）
    glm::vec2 aabb_half_extents(two.Size.x / 2, two.Size.y / 2);
    glm::vec2 aabb_center(
        two.Position.x + aabb_half_extents.x,
        two.Position.y + aabb_half_extents.y
    );
    // 获取两个中心的差矢量
    glm::vec2 difference = center - aabb_center;
    glm::vec2 clamped = glm::clamp(difference, -aabb_half_extents,
aabb_half_extents);
    // AABB_center加上clamped这样就得到了碰撞箱上距离圆最近的点closest
    glm::vec2 closest = aabb_center + clamped;
    // 获得圆心center和最近点closest的矢量并判断是否 length <= radius
    difference = closest - center;
    return glm::length(difference) < one.Radius;
}
```

我们创建了CheckCollision的一个重载函数用于专门处理一个BallObject和一个GameObject的情况。因为我们并没有在对象中保存碰撞外形的信息，因此我们必须为其计算：首先计算球心，然后是AABB的半边长及中心。

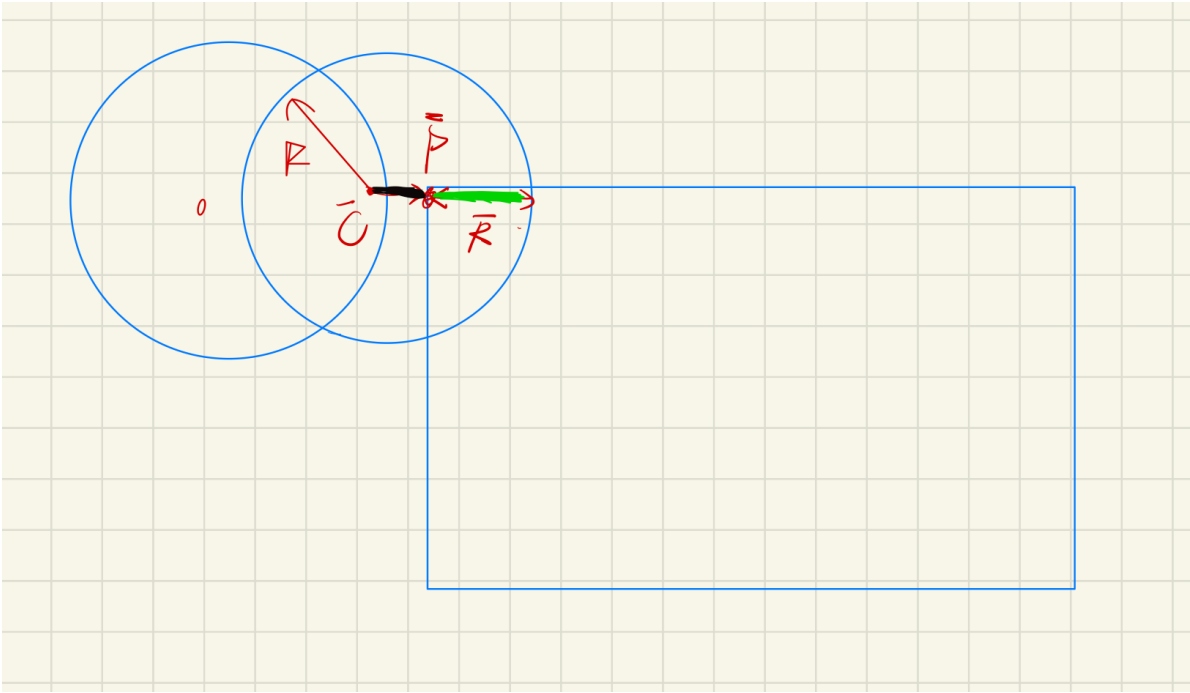
使用这些碰撞外形的参数，我们计算出differenceD¯D¯然后得到限制后的值clamped，并与AABB中心相加得到closestP¯P¯。然后计算出center和closest的矢量差D'¯D'¯并返回两个外形是否碰撞。

之前我们调用CheckCollision时将球对象作为其第一个参数，因此现在CheckCollision的重载变量会自动生效，我们无需修改任何代码。

六：碰撞处理

1.碰撞重新定位

目的是为了纠正球的位置，使得球在碰撞时的效果是两者刚好接触，没有重叠的效果。这里难就难在如何定位。在第五部分已经说明如何检测碰撞，已经求的最近距离。现在我们来校正位置。



此时球少量进入了长方形，所以检测到了碰撞。我们现在希望将球从移出长方形的外形使其仅仅碰触到长方形，像是没有碰撞一样。为了确定需要将球从长方形中移出多少距离，我们需要找回矢量R¯R¯，它代表的是侵入长方形的程度。为得到R¯我们用球的半径减去V¯。矢量V¯是最近点P¯和球心C¯的差矢量。

有了R¯之后我们将球的位置偏移R¯就将球直接放置在与长方形紧邻的位置；此时球已经被重定位到合适的位置。

2.确定反弹方向

下一步我们需要确定碰撞之后如何更新球的速度。我们使用以下规则来改变球的速度：

1. 如果球撞击长方体的右侧或左侧，它的水平速度（x）将会反转。
2. 如果球撞击长方体的上侧或下侧，它的垂直速度（y）将会反转。

在opengl的变换教程中，有谈到关于夹角变换的问题，这里也是用到同样原理。

如果我们定义指向北、南、西和东的四个矢量，然后计算它们和给定矢量的夹角

```
Direction VectorDirection(glm::vec2 target)
{
    glm::vec2 compass[] = {
        glm::vec2(0.0f, 1.0f),  // 上
        glm::vec2(1.0f, 0.0f),  // 右
        glm::vec2(0.0f, -1.0f), // 下
        glm::vec2(-1.0f, 0.0f)  // 左
```

```
    };
    GLfloat max = 0.0f;
    GLuint best_match = -1;
    for (GLuint i = 0; i < 4; i++)
    {
        GLfloat dot_product = glm::dot(glm::normalize(target), compass[i]);
        if (dot_product > max)
        {
            max = dot_product;
            best_match = i;
        }
    }
    return (Direction)best_match;
}
```

此函数比较了target矢量和compass数组中各方向矢量。compass数组中与target角度最接近的矢量，即是返回给函数调用者的Direction。这里的Direction是一个Game类的头文件中定义的枚举类型：

```
enum Direction {
    UP,
    RIGHT,
    DOWN,
    LEFT
};
```

3.球块碰撞

为计算碰撞处理所需的数值，我们需要更多的数值而不是ture or false,还需返回碰撞发生时的方向及差矢量（$\bar{R}$）

为了更好组织代码，我们把碰撞相关的数据使用typedef定义为Collision

```
typedef std::tuple<GLboolean, Direction, glm::vec2> Collision;
```

接下来我们还需要修改CheckCollision函数的代码，使其不仅仅返回true或false而是还包含方向和差矢量：

Game类的DoCollision函数现在不仅仅只检测是否出现了碰撞，而且在碰撞发生时会有适当的动作。此函数现在会计算碰撞侵入的程度并且基于碰撞方向使球的位置矢量与其相加或相减。

。首先我们会检测碰撞如果发生了碰撞且砖块不是实心的那么就销毁砖块。然后我们从tuple中获取到了碰撞的方向dir以及表示$\bar{V}$的差矢量diff_vector，最终完成碰撞处理。

我们首先检查碰撞方向是水平还是垂直，并据此反转速度。如果是水平方向，我们从diff_vector的x分量计算侵入量R并根据碰撞方向用球的位置矢量加上或减去它。垂直方向的碰撞也是如此，但是我们要操作各矢量的y分量。

4.玩家球碰撞

机制：基于撞击挡板的点与（挡板）中心的距离来改变球的水平速度。撞击点距离挡板的中心点越远，则水平方向的速度就会越大。

```
void Game::DoCollisions()
{
    [...]
    Collision result = CheckCollision(*Ball, *Player);
```

```
    if (!Ball->Stuck && std::get<0>(result))
    {
        // 检查碰到了挡板的哪个位置，并根据碰到哪个位置来改变速度
        GLfloat centerBoard = Player->Position.x + Player->Size.x / 2;
        GLfloat distance = (Ball->Position.x + Ball->Radius) - centerBoard;
        GLfloat percentage = distance / (Player->Size.x / 2);
        // 依据结果移动
        GLfloat strength = 2.0f;
        glm::vec2 oldVelocity = Ball->Velocity;
        Ball->Velocity.x = INITIAL_BALL_VELOCITY.x * percentage * strength;
        Ball->Velocity.y = -Ball->Velocity.y;
        Ball->Velocity = glm::normalize(Ball->Velocity) *
glm::length(oldVelocity);
    }
}
```

5.粘板

运行代码时，球和玩家挡板的碰撞处理仍旧有一个大问题，原因是玩家挡板以较高的速度移向球，导致球的中心进入玩家挡板。由于我们没有考虑球的中心在AABB内部的情况，游戏会持续试图对所有的碰撞做出响应，当球最终脱离时，已经对`y`向速度翻转了多次，以至于无法确定球在脱离后是向上还是向下运动

我们可以引入一个小的特殊处理来很容易地修复这种行为，这个处理之所以成为可能是基于我们可以假设碰撞总是发生在挡板顶部的事实。我们总是简单地返回正的y速度而不是反转y速度，这样当它被卡住时也可以立即脱离

```
 //Ball->Velocity.y = -Ball->Velocity.y;
Ball->Velocity.y = -1 * abs(Ball->Velocity.y);
```

6.底部边界

在Game类的Update函数中，我们要检查球是否接触到了底部边界，接触3次以上终止游戏。

```
void Game::Update(GLfloat dt)
{
    [...]
    if (Ball->Position.y >= this->Height) // 球是否接触底部边界？
    {
        this->ResetLevel();
        this->ResetPlayer();
    }
}
```

七：粒子效果

这个花里胡哨的东西就不讲怎么实现了，自己可以查阅opengl相关文档。具体效果就是让球带上一串“尾气”一样的东西。

代码在 particle_generator.h

八：功能加强

可以通过几个后期处理特效丰富游戏的视觉效果

们的后期处理着色器允许使用三种特效：**shake**, **confuse**和**chaos**。

- **shake**：轻微晃动场景并附加一个微小的模糊效果。
- confuse：反转场景中的颜色并颠倒x轴和y轴。
- **chaos**: 利用边缘检测卷积核创造有趣的视觉效果，并以圆形旋转动画的形式移动纹理图片，实现“混沌”特效。

以下是这些效果的示例：

这些功能都是opengl里面的内容，与程序设计没太大关系，不详细讲述原理。可查阅帧缓冲与抗锯齿与着色器的相关内内容

除此以外：作为这些效果的演示，我们将模拟球击中坚固的混凝土块时的视觉冲击。无论在哪里发生碰撞，只要在短时间内实现晃动(shake)效果，便能增强撞击的冲击感

九：道具

为了让游戏更加好玩

无论一个砖块何时被摧毁，它都有一定几率产生一个道具块。这样的道具快会缓慢降落，而且当它与玩家挡板发生接触时，会发生基于道具类型的有趣效果。例如，某一种道具可以让玩家挡板变长，另一种道具则可以让小球穿过物体。我们还可以添加一些可以给玩家造成负面影响的负面道具。

我们可以在Gameobject添加额外的成员属性

每个道具以字符串的形式定义它的类型，持有表示它有效时长的持续时间与表示当前是否被激活的属性

- **Speed**: 增加小球20%的速度
- **Sticky**: 当小球与玩家挡板接触时，小球会保持粘在挡板上的状态直到再次按下空格键，这可以让玩家在释放小球前找到更合适的位置
- **Pass-Through**: 非实心砖块的碰撞处理被禁用，使小球可以穿过并摧毁多个砖块
- **Pad-Size-Increase**: 增加玩家挡板50像素的宽度
- **Confuse**: 短时间内激活confuse后期特效，迷惑玩家　//第八节　玩家应避开
- **Chaos**: 短时间内激活chaos后期特效，使玩家迷失方向　//第八节　玩家应该避开

每次砖块被摧毁时我们希望以一定几率（1/75普通道具，1/15负面道具）生成一个道具，这个功能可以在Game的SpawnPowerUps函数中找到：

十：音效

碰撞一次发出声音，背景音乐循环放

OpenGL不提供关于音频的任何支持。我们将使用被称为irrKlang的音频管理库

[solid.wav]：小球撞击实心砖块时的音效