

## 华东师范大学软件学院实验报告

实验课程：OOAD

年级：2020 级

实验成绩：

实验名称：GizmoBall

姓名：

实验编号：No.3 弹球游戏

10192100578 任璐

实验日期：2022.10.26

指导教师：姜宁康

10205101530 赵晗瑜

实验时间：4 学时

### 一、实验目的

1. 设计并实现弹球游戏（类似三维弹球）
2. 完成相应文档和演示
3. 三人一组

### 二、实验内容与实验要求

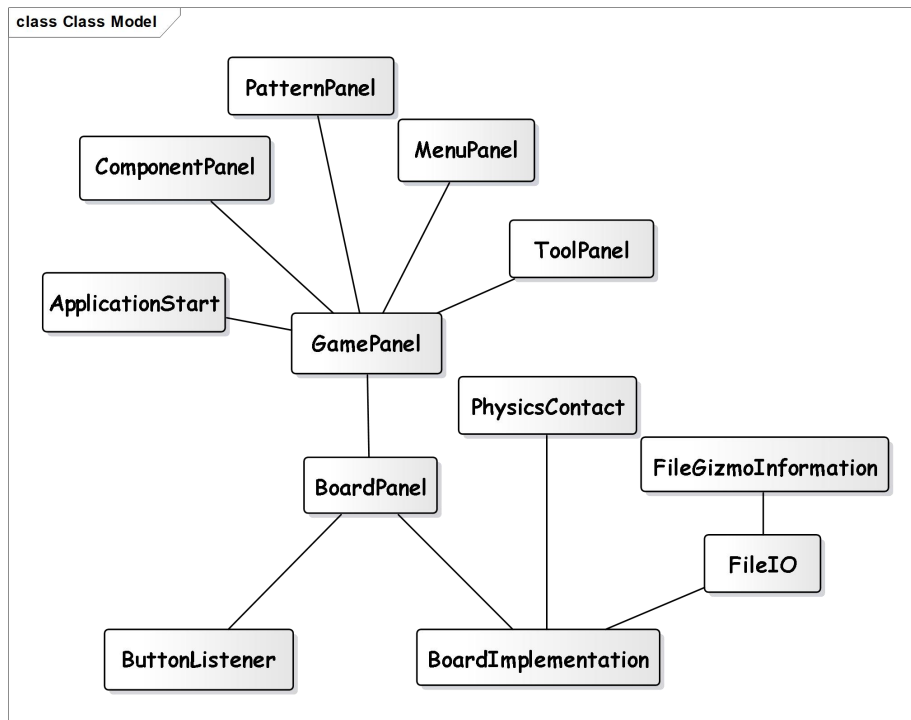
代码仓库：[hot-zhy/ecnu ooad gizmoball at master \(github.com\)](https://github.com/hot-zhy/ecnu_ooad_gizmoball_at_master)

#### 1. 实现思路

- 1) 为了实现物理世界的模拟碰撞效果，采用 Jbox2D 开源物理引擎，设定重力、密度、摩擦系数和弹性系数来进行 2D 刚体物理运动的模拟；
- 2) 前端实现部分，将游戏面板 GamePanel 分为四个部分组件面板（ComponentPanel）、模式面板（PatternPanel）、工具栏面板（ToolPanel）、菜单面板（MenuPanel），将 GamePanel 类作为游戏的主控，控制主窗口信息的绘制以及四个组件面板的绘制、以及一些初始化工作；
- 3) 利用 GRASP 的外观控制器原则，设计一个 BoardPanel 类来监听响应用户的点击和绘制，实现游戏棋盘静态界面的绘制，并将用户在界面操作的信息（包括放置组件和使用工具）传递给后端，控制线程刷新；
- 4) 后端实现部分，设计一个 PhysicsContact 类来实现物理世界的创建和设置，以及完成对物理世界组件的添加以及碰撞的检测；设计一个 BoardImplementation 类将用户点击添加的物体加入其列表，并将这些物体在窗口画出来（游玩模式和设计模式的绘制），实现物理世界物体的放大、缩小、旋转和删除，并将物体添加到物理世界来模拟碰撞；

5) 设计一个启动类 `ApplicationStart` 新建游戏面板打开窗口。

## 2. 领域模型设计

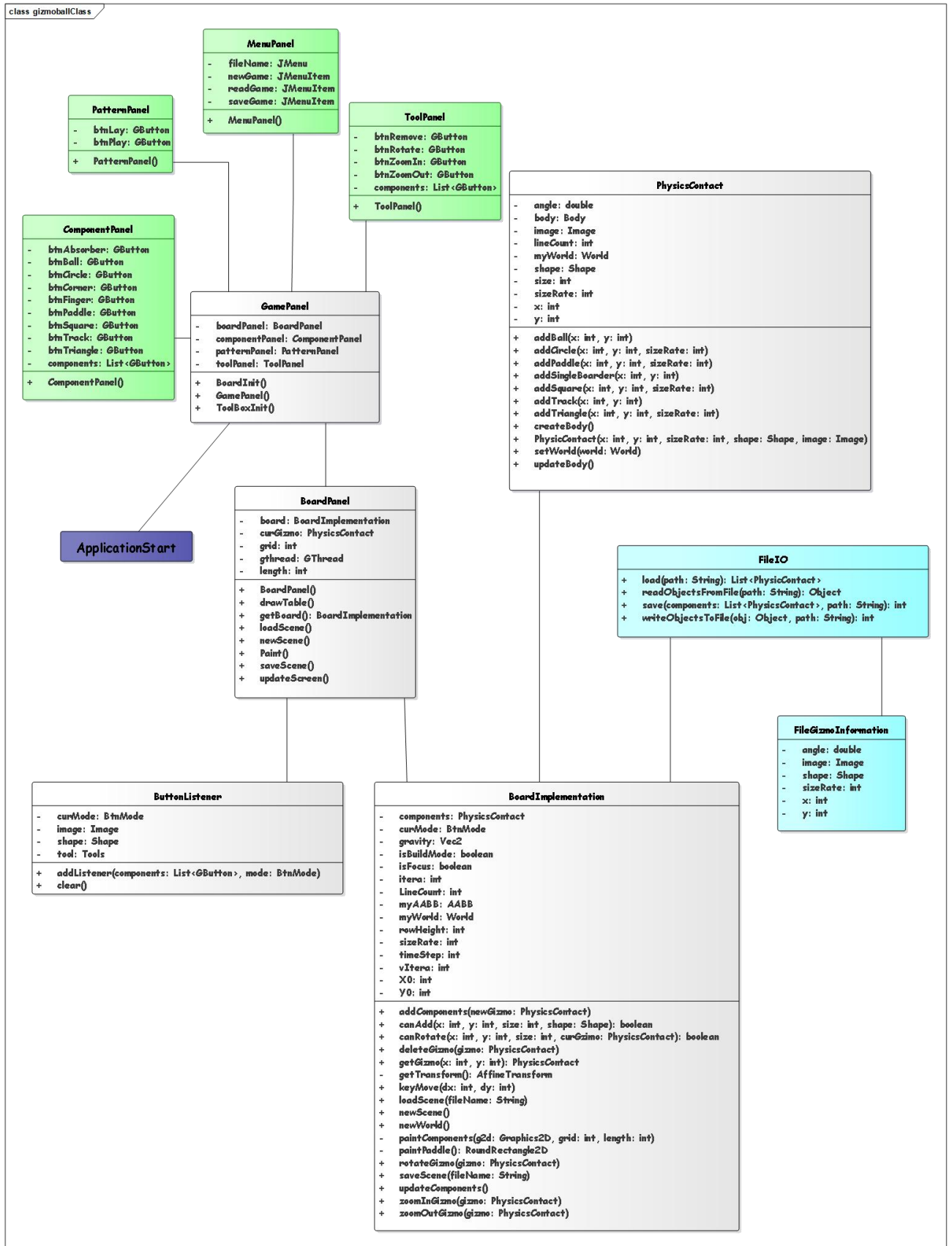


## 3. 类图设计

职责分配：

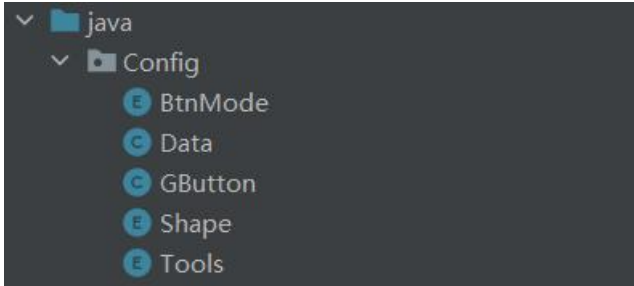
类	职责
<b>ApplicationStart</b>	游戏控制的主入口
<b>GamePanel</b>	作为游戏 UI 的主控，控制主窗口信息的绘制以及四个组件面板的绘制、以及一些初始化工作
<b>ComponentPanel</b>	实现游戏界面右侧组件栏的绘制及其各个组件对应的功能
<b>PatternPanel</b>	实现游戏界面右侧模式栏的绘制及其两个按钮对应的功能
<b>MenuPanel</b>	实现游戏上方菜单栏的对应功能，包括“新建游戏”、“保存游戏”和“读取游戏”
<b>ToolPanel</b>	实现游戏界面右侧工具栏的绘制及其各个工具

	对应的功能
<b>BoardPanel</b>	实现游戏棋盘界面的静态绘制（布局模式下的棋盘界面）。包括背景网格的绘制、组件的摆放以及组件的旋转、放大、缩小、删除的界面显示效果。负责将用户在界面操作的信息（前端各个 Panel 的信息）传递给后端 <b>BoardImplmentation</b> 类并控制线程刷新。
<b>ButtonListener</b>	实现右侧工具栏和组件栏按钮的监听功能
<b>BoardImplementation</b>	保存游戏棋盘中的物理信息列表，绘制用户添加过的物体组件，并为 <b>gizmo</b> 游戏设置物理游戏来模拟物体碰撞，同时实现刚体组件的放大、缩小、旋转和删除，实现布局模式
<b>PhysicsContact</b>	在物理世界中添加各类刚体组件，监听物理世界中刚体的碰撞并模拟碰撞效果，并实现当放大、缩小、旋转物体时更新物理刚体的功能
<b>FileIO</b>	实现游戏中组件文件的写入、从文件中加载物理组件并写入棋盘的功能
<b>FileGizmoInformation</b>	简化和提取要写入文件的组件的属性信息



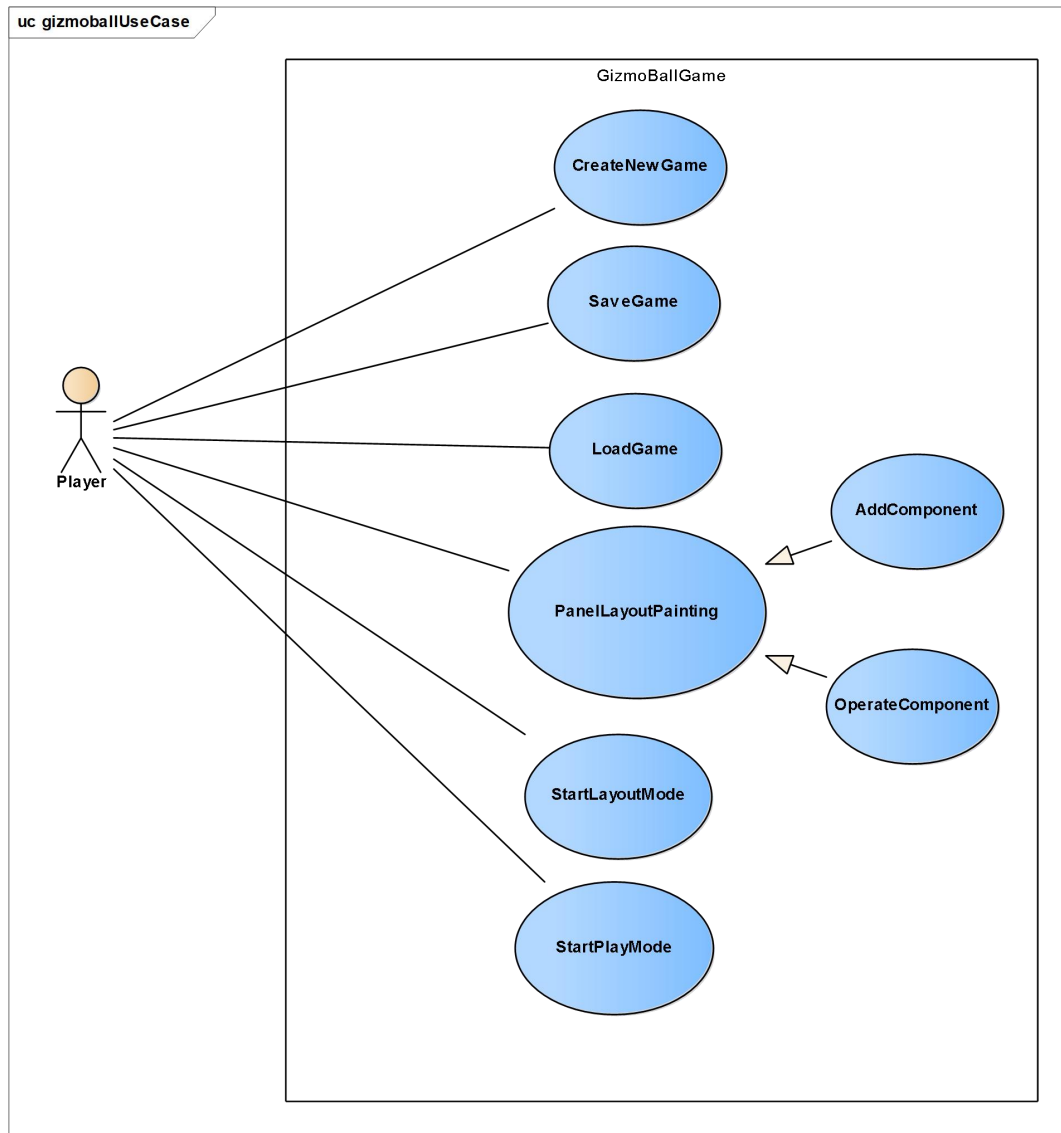
#### 4. 类图设计中的职责分配原则

<p><b>Information Expert</b></p>	<p>在设计的过程中一直在考虑信息专家原则</p> <p>【问】应该由哪个类负责绘制右侧的组件栏的各个物体图片、工具栏的各个图片以及模式栏按钮等信息？</p> <p>【答】应该由 <b>GamePanel</b> 类来负责，因为该类具有能够完成其绘制的所有信息，包括组件栏、工具栏和模式栏中的所有图片列表等信息。</p> <p>【问】应该由哪个类负责碰撞检测？</p> <p>【答】应该由 <b>PhysicsContact</b> 类来负责，因为该类具有能够完成其绘制的所有信息，包括用户绘制的物体列表以及 <b>jbox2d</b> 库的所有信息。</p>
<p><b>Creator</b></p>	<p>【问】应该由哪个类来创建 <b>BoardImplementation</b> 的实例？</p> <p>【答】应该由 <b>BoardPanel</b> 类来创建，因为 <b>BoardPanel</b> 频繁使用 <b>BoardImplementation</b>，<b>BoardPanel</b> 监听得到的用户绘制的物体，都需要传给 <b>BoardImplementation</b> 类来进行绘制和碰撞的检测，因此 <b>BoardImplementation</b> 的实例由 <b>BoardPanel</b> 类来创建。</p> <p>【问】应该由哪个类来创建 <b>ComponentPanel</b> 类的实例？</p> <p>【答】应该由 <b>GamePanel</b> 类来创建，因为 <b>ComponentPanel</b> 的实例是 <b>GamePanel</b> 实例的一部分，<b>GamePanel</b> 记录着 <b>ComponentPanel</b> 的实例</p>
<p><b>Low coupling</b></p>	<p><b>BoardPanel</b> 和 <b>PhysicsContact</b> 之间的关联通过 <b>BoardImplementation</b> 类来实现，并不是将两者直接相连（这样会增加耦合度），这样的设计符合低耦合原则。</p>
<p><b>High cohesion</b></p>	<p>将功能相关的模块集合在一起，如在设计中，将绘制主窗口和组件栏、工具栏、模式栏等相关功能分配给 <b>GamePanel</b> 类；</p> <p>将添加刚体和模拟刚体碰撞等相关功能分配给 <b>PhysicsContact</b> 类；</p> <p>将绘制游戏各类物体组件分配给 <b>BoardPanel</b> 类；</p> <p>将各个按钮监听功能分配给 <b>ButtonListener</b> 类；</p> <p>每个模块都尽可能完成自己的功能，符合高内聚原则。</p>
<p><b>Controller</b></p>	<p><b>BoardPanel</b> 用来监听用户对物体和工具按钮的选择，该类用来接收和处理系统事件，该类作为外观控制器，连接着 UI 层和内部实现系统。</p>

<b>Indirection</b>	<p>为了避免各个各个按钮（GButton）直接和 BoardPanel 和 BoardImplementation 直接耦合，设计一个按钮监听器类（ButtonListener）来作为中介，降低耦合性，该中介实现了 GButton 和 BoardPanel 之间以及 GButton 和 BoardImplementation 之间的间接性（Indirection）。</p>
<b>Protected variations</b>	<p>以数据驱动设计为导向，进行数据封装，将 Data 类中的图片数据、GButton 中的按钮数据、Shape 中的类型数据、Tools 中的工具数据封装起来，防止此类数据的变化对系统产生影响，便于修改。</p> 

## 5. 用例图

参与者 <b>Player</b>	玩游戏的玩家
用例 <b>CreateNewGame</b>	新建场景，新建一场游戏
用例 <b>SaveGame</b>	保存当前面板中的组件信息到文件
用例 <b>LoadGame</b>	从文件中载入当前游戏中的组件信息到面板
用例 <b>PanelLayoutPainting</b>	在面板上绘制
用例 <b>StartLayoutModel</b>	点击开始布局按钮开始布局
用例 <b>StartPlayMode</b>	点击开始游玩按钮开始游玩
用例 <b>AddComponent</b>	布局模式下添加组件
用例 <b>OperateComponent</b>	布局模式下对组件进行放大、缩小、旋转和删除



## 6. 用例描述

### 1) 用例 PanelLayoutPainting

<b>Section Title</b>	PanelLayoutPainting
<b>Summary</b>	The use case allows a Player to paint t,including add components on the panel and zoom in、zoom out、rotate、delete it.
<b>Actor List</b>	Player
<b>Pre-condition</b>	The panel was initialized normally and layout mode has been turned on.
<b>Description</b>	<ol style="list-style-type: none"> <li>1. The Player clicks on the component to be added in the component bar.</li> <li>2. The player clicks the position on the board panel where he wants to add the component he has chosen.</li> </ol>

	<p>3. The player click the tool button which he wants to use on the component.</p> <p>4. The player clicks the component on the board panel which he wants to operate using the tool he has chosen.</p> <p>5. Repeat step 1-4,until layout painting is finished.</p>
<b>Post-condition</b>	The Player finishes drawing the panel, and the system saves the information of the components on the panel.
<b>Exception</b>	<p>2a There is already a component at the position clicked by the user</p> <p>2a.1 The system prompts the warning and player fails to add compoment</p> <p>4a The selected component cannot be rotated</p> <p>4a.1 The system prompts the warning and the player fails to rotate the component.</p>

## 2) 用例 StartPlayMode

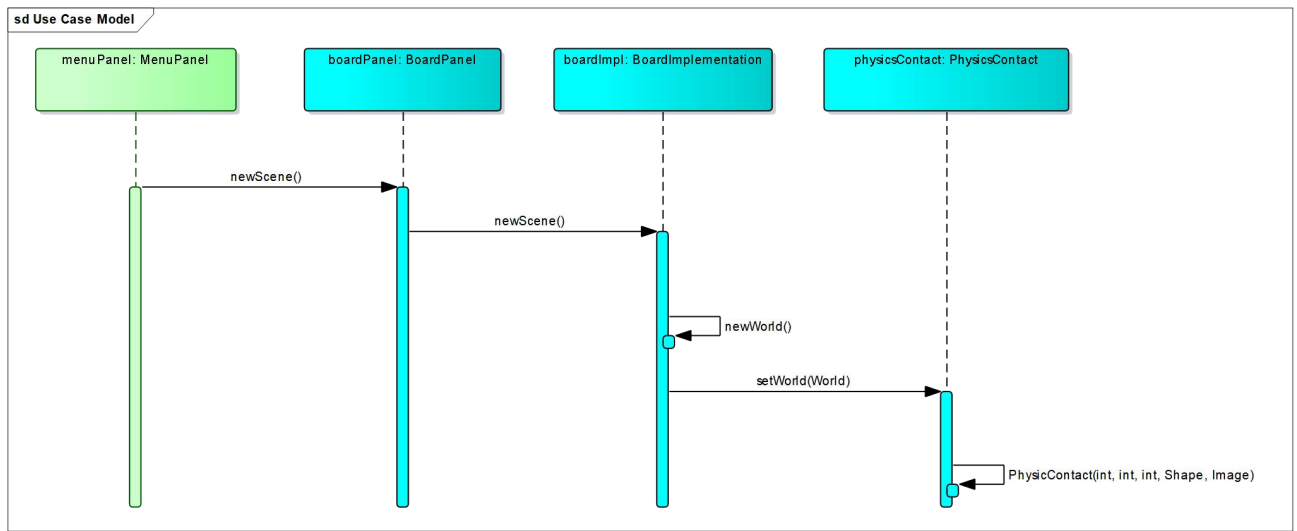
<b>Section Title</b>	StartPlayMode
<b>Summary</b>	This use case anutomates the simulation of collisions and uses a physics engine and allows the paddle to be moved through the keyboard.
<b>Actor List</b>	Player
<b>Pre-condition</b>	The played has add ball component on the board in the layout mode.
<b>Description</b>	<p>1. The player clicks the button to start the game mode,and the physics engines start to simulate automatically.</p> <p>2. The player uses the up、down、left and right keys on the keyboard to control the paddle to move.</p> <p>3. The system continues to simulate the game until the ball touches the absorber or the low boundary, the ball disappears and the game is over.</p>
<b>Post-condition</b>	The physics engine can simulate normally,the ball is absorbed finally and the game is over normally.
<b>Exception</b>	<p>2a There is no paddle on the board.</p> <p>2a.1 There is no effect pressing the keyboard.</p> <p>3a The ball cannot touch the absorber or the low boundary</p>



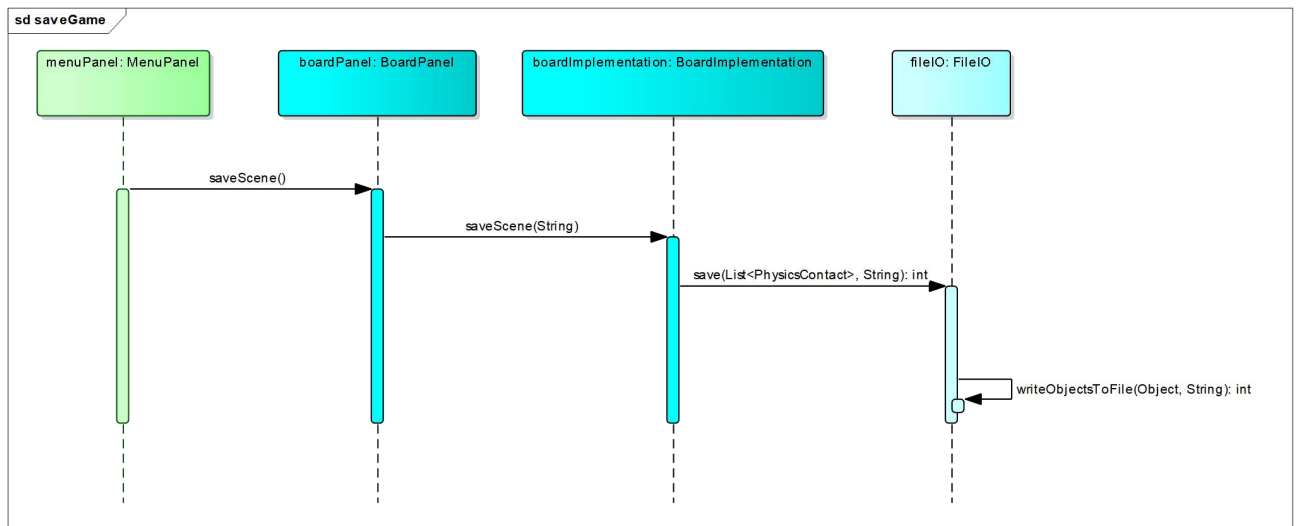
	<p>3a.1 The ball will continues to do a certain movement and cannot stop</p> <p>2b The player presses the keyboard too fast</p> <p>2b.1 The system cannot accurately read every keyboard keypress.</p>
--	--

## 7. 重点过程顺序图

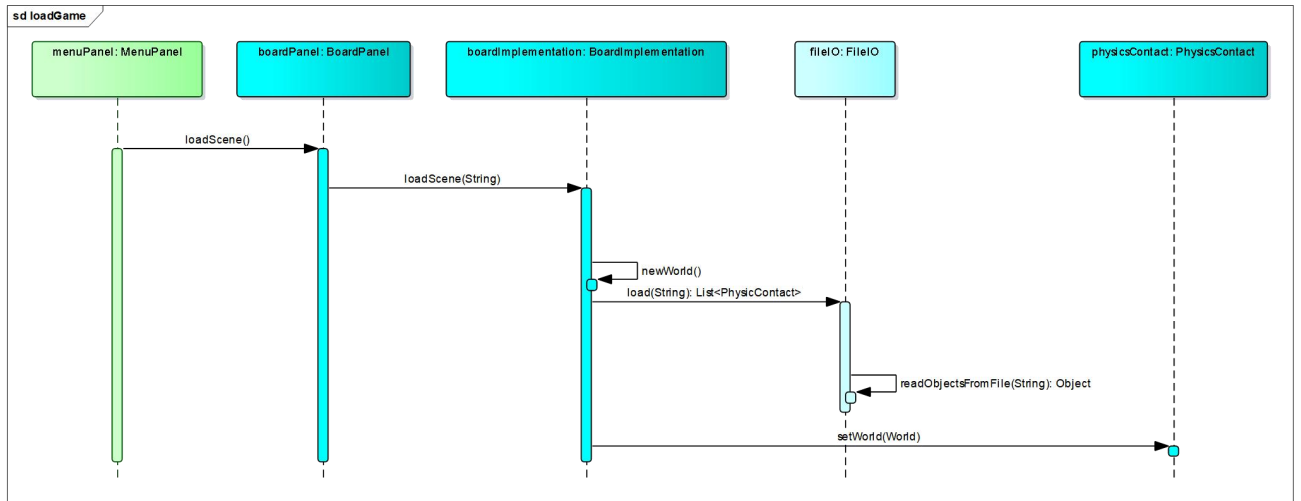
### 1) 新建游戏



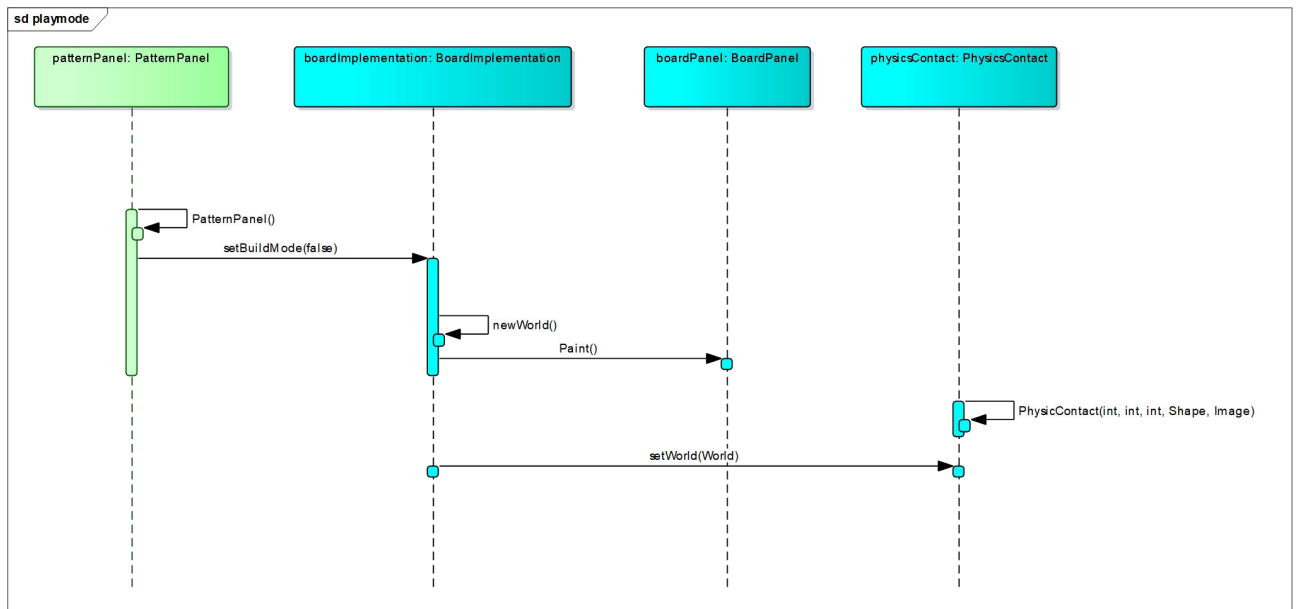
### 2) 保存游戏



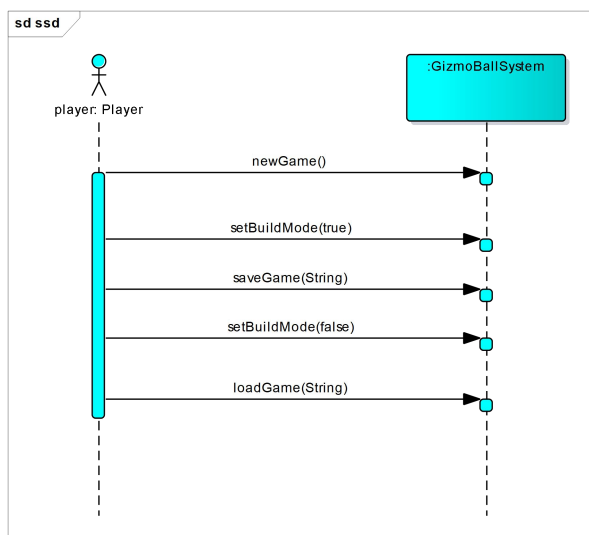
### 3) 读取游戏



#### 4) 开启游玩模式



### 8. 系统顺序图



## 9. 问题难点解决

### 1) 如何根据物体类型创建物理世界中的物体,并设置物体的形状?

在 **PhysicsContact** 类中定义一个 **CreateBody** 函数, 判断刚体类型进行创建:

```

1.      /**
2.       * 创建刚体组件
3.       */
4.     public void createBody(){
5.         //判断要创建刚体的类型
6.         switch (shape){
7.             case Ball:
8.                 addBall(x, y);
9.                 break;
10.            case Absorber:
11.                addSquare(x, y, sizeRate);
12.                break;
13.            case Triangle:
14.                addTriangle(x, y, sizeRate);
15.                break;
16.            case Circle:
17.                addCircle(x, y, sizeRate);
18.                break;
19.            case Square:
20.                addSquare(x, y, sizeRate);
21.                break;
22.            case Paddle:
23.                addPaddle(x, y, sizeRate);
24.                break;
25.            case Track:
26.                addTrack(x, y);
27.                break;
28.            case Corner:
29.                addSquare(x, y, sizeRate);
30.                break;
31.        }
32.    }
33.    //设置用户数据
34.    body.setUserData(shape);
35. }
```

以添加球体为例, 首先添加一个刚体的描述, 对刚体的信息进行设置(球体), 然后将刚体描述挂载在创建的刚体上, 同时为球刚体设置形状、密度、弹性系数和初始线性速度等信息:

```

1.      /**
2.       * 添加刚体球
3.       * @param x
4.       * @param y
5.       */
```

```

6.     public void addBall(int x,int y){
7.         Data.ballCount++;
8.         y = 25 - y;
9.         float r = size / 2.0f;
10.
11.        //新建刚体描述
12.        BodyDef bd=new BodyDef();
13.        //设置刚体类型为动态
14.        bd.type = BodyType.DYNAMIC;
15.        //设置刚体位置
16.        bd.position=new Vec2((x) * size + r / 2.0f, (y) * size - r / 2.0f);
17.        //允许休眠
18.        bd.allowSleep=true;
19.        //为刚体挂载描述信息
20.        body = myWorld.createBody(bd);
21.
22.
23.        CircleShape circleShape = new CircleShape();
24.        circleShape.setRadius(r/1.5f);
25.        //新建刚体物理描述,描述摩擦系数、弹性系数、密度
26.        FixtureDef fixtureDef = new FixtureDef();
27.        //设置形状
28.        fixtureDef.shape = circleShape;
29.        //设置密度
30.        fixtureDef.density = 1;
31.        //设置弹性系数
32.        fixtureDef.restitution = 1;
33.        body.createFixture(fixtureDef);
34.        body.setLinearVelocity(new Vec2(0,-10000));
35.        body.setGravityScale(10);
36.    }

```

其中要注意的是，**resitution** 属性是弹性系数（补偿系数），弹性系数为 0 时不会回弹，为 1 时会回弹（弹性碰撞）

## 2) 如何监听和控制游玩模式中各个物体间的碰撞？

利用 **Jbox2d World** 类中的 **ContactListener** 监听器，重写 **beginContact** 方法监听物理世界中物体的碰撞

```

1.     public static void setWorld(World world) {
2.         PhysicsContact.myWorld = world;
3.         PhysicsContact.LINECOUNT = BoardImplementation.getLineCount();
4.        //注册监听碰撞器
5.        myWorld.setContactListener(new ContactListener() {
6.            //监听碰撞
7.            @Override
8.            public void beginContact(Contact contact) {
9.                //碰撞物体
10.                Body body1 = contact.getFixtureA().getBody();
11.                //被碰撞物体

```

```
12.         Body body2 = contact.getFixtureB().getBody();
13.         // 被碰撞物体也是球，完全弹性碰撞，二者交换
14.         if (body2.getUserData() == Shape.Ball) {
15.             Body b = body1;
16.             body1 = body2;
17.             body2 = b;
18.         }
19.         // 碰撞物体是球
20.         if (body1.getUserData() == Shape.Ball) {
21.             // 被碰撞物体是漩涡黑洞
22.             if (body2.getUserData() == Shape.Absorber) {
23.                 // 将球的信息销毁
24.                 body1.setUserData(null);
25.                 Data.ballCount--;
26.                 // 在物理世界中销毁球
27.                 myWorld.destroyBody(body1);
28.             }
29.             // 碰到弯轨改变小球速度
30.             else if (body2.getUserData() == Shape.Corner) {
31.                 body1.setLinearVelocity(new Vec2(-10000, 0));
32.             }
33.             // 碰到下边界销毁小球
34.             else if (body2.getUserData() == Shape.Bound) {
35.                 body1.setUserData(null);
36.                 Data.ballCount--;
37.                 myWorld.destroyBody(body1);
38.             }
39.         }
40.     }
41. });
42. }
```

### 3) 如何实现游玩模式下界面的刷新和绘制？

在 **BoardPanel** 类中扩展 **Thread** 类重写 **run** 方法，控制游玩模式下前端面板的刷新：

```
1.     class GThread extends Thread {
2.         @Override
3.         public void run() {
4.             while(!isBuildMode()) {
5.                 // 调用 BoardImplementation 中的 setStep 方法
6.                 board.setStep();
7.                 // 更新面板
8.                 updateScreen();
9.                 try {
10.                     Thread.sleep((long) (getTimeStep() * 1000));
11.                 } catch (InterruptedException e) {
12.                     break;
13.                 }
14.             }
15.             // 更新面板
```

```

16.         updateScreen();
17.     }
18. }

```

**BoardImplementation** 中的 **setStep** 方法:

```

1.     public void setStep(){
2.         myWorld.step(timeStep, vItera, itera);
3.     }

```

调用了 **jbox2d** 中的 **step** 函数:

```

void step ( float timeStep, int
velocityIterations, int position
Iterations)

```

进行一轮迭代模拟, **timeStep** 为模拟的时间步进, **velocityIterations** 为速度迭代次数, **positionIterations** 为位置迭代次数

采用 **build** 方法进行线程的总控:

```

1.     public void build() {
2.         if(isBuildMode()) {
3.             gThread.interrupt();
4.             board.updateComponents();
5.         }
6.         else {
7.             gThread = new GThread();
8.             gThread.start();
9.             this.requestFocus();
10.        }
11.    }

```

#### 4) 如何实现放大/缩小/旋转/删除物体?

在实现旋转、放大等操作时, 现将原来的物体从物理世界中删除(**deleteBody**), 在保存原数据的基础上进行旋转和实现缩放, 再重新创建新的刚体放入物理世界(**createBody**)

```

1.     /**
2.      * 放大、缩小、旋转物体时, 首先销毁原来的物体, 再重新创新新的改变后的物体
3.      */
4.     public void updateBody() {
5.         myWorld.destroyBody(body);
6.         createBody();
7.     }

```

在对物体进行旋转之前, 首先应判断该物体是否可旋转

```

1.     public boolean canRotate(int x, int y, int size, PhysicsContact curGizmo){
2.         int dx, dy;
3.         for (PhysicsContact giz : components){
4.             dx = giz.getX();
5.             dy = giz.getY();

```

```

6.         if (dx >= x && dy <= y && dx <= x + size && dy >= y -
            size && giz.getShape() != curGizmo.getShape()){
7.             return false;
8.         }
9.     }
10.    return true;
11. }

```

然后再对物体进行旋转 90 度：

```

1.    public void rotateGizmo(PhysicsContact gizmo){
2.        if (gizmo != null) {
3.            gizmo.setAngle(gizmo.getAngle() + Math.PI / 2);
4.            gizmo.updateBody();
5.        }
6.    }

```

放大与旋转类似，首先判断能否放大（是否与周围的物体冲突），然后再放大，其中实现放大缩小使用了 **sizeRate**（缩放比例）属性，对于放大则将 **sizeRate** 加一，对于缩小则将 **sizeRate** 减一，当 **sizeRate** 为 0 时物体会消失。

删除物体时，首先调用 **destroyBody(gizmo.getBody())**消除物理世界中的物体，然后调用 **remove** 移出组件列表中的该物体。

## 5) 如何实现文件的保存和读取？

在 **BoardImplementation** 类中维护一个 **components** 列表保存物理刚体，每添加一个物体到物理世界便将其添加到该列表：

```

1.    public void addComponents(PhysicsContact newGizmo){
2.        components.add(newGizmo);
3.    }

```

新建游戏即将文件列表清空：

```

1.    public void newScene() {
2.        newWorld();
3.        components.clear();
4.    }

```

保存游戏即在 **BoardImplementation** 类中调用 **FileIO** 类中保存（save）组件信息的方法：

```

1.    public void saveScene(String fileName) {
2.        FileIO.save(components, fileName);
3.    }

```

**save** 函数：

```

1.    //将当前游戏面板中所有组件写入文件
2.    public static void save(List<PhysicsContact> components, String path) {
3.        List<FileGizmoInformation> list = new ArrayList<>(); //创建存入文件的组件列表
4.        //遍历传入的列表，将其转换为 FileGizmoInformation 形式加入 list
5.        for(PhysicsContact i : components) {

```

```

6.         FileGizmoInformation temp = new FileGizmoInformation();
7.         temp.setX(i.getX());
8.         temp.setY(i.getY());
9.         temp.setAngle(i.getAngle());
10.        temp.setSizeRate(i.getSizeRate());
11.        temp.setShape(i.getShape());
12.        list.add(temp);
13.    }
14.    writeObjectToFile(list, path); //将List 信息存入文件
15. }

```

为什么需要一个 **FileGizmoInformation** 类？该类的主要职责是将组件中需要保存的属性信息提取和简化出来，便于文件的存入和读取。

读取游戏即在 **BoardImplementation** 类中调用 **FileIO** 类中的 **load** 方法：

```

1.     public void loadScene(String fileName) {
2.         World nowWorld = myWorld;
3.         newWorld();
4.         List<PhysicsContact> list = FileIO.load(fileName);
5.         if (components == null) {
6.             myWorld = nowWorld;
7.         } else {
8.             components = list;
9.         }
10.    }

```

**load** 函数：

```

1.     //从文件中载入地图
2.     public static List<PhysicsContact> load(String path) {
3.         List<FileGizmoInformation> list = (List<FileGizmoInformation>) readObjectFromFile(path); //从
        文件中读入 FileGizmo 类型的组件信息
4.         List<PhysicsContact> components = new ArrayList<>(); //创建 Gizmo 类型列表
5.         for(FileGizmoInformation i : list) {
6.             //根据类型设置组件图片样式
7.             switch (i.getShape()) {
8.                 case Ball:
9.                     i.setImage(ballIcon.getImage());
10.                    break;
11.                 case Circle:
12.                     i.setImage(circleIcon.getImage());
13.                    break;
14.                 case Track:
15.                     i.setImage(trackIcon.getImage());
16.                    break;
17.                 case Paddle:
18.                     i.setImage(paddleIcon.getImage());
19.                    break;
20.                 case Square:
21.                     i.setImage(squareIcon.getImage());
22.                    break;
23.                 case Absorber:

```



```
24.         i.setImage(absorberIcon.getImage());
25.         break;
26.         case Triangle:
27.             i.setImage(triangleIcon.getImage());
28.             break;
29.         default:
30.             }
31.         PhysicsContact gizmo = new PhysicsContact(i.getX(),i.getY(),i.getSizeRate(),i.getShape(),i.
getImage());
32.         // 设置角度
33.         if(i.getAngle() != 0) {
34.             gizmo.setAngle(i.getAngle());
35.             gizmo.updateBody();
36.         }
37.         components.add(gizmo);
38.     }
39.     return components;
40. }
```

## 10. 总结与体会

- 1) 本次实验的最大难点在于如何将物理引擎 **jbox2d** 引入系统，之所以选择使用 **jbox2d** 是本着降低开发成本的目的，然而其没有官方文档，只能下载阅读其源码以及阅读网络博客了解使用其 **API**，因此我们提高了自己的阅读源码能力和学习能力；
- 2) 体会了各种 **GRASP** 原则的应用，对理论课的知识掌握进一步加深；
- 3) 学习了文件系统相关的知识，用 **List** 记录 **PhysicsContact** 的信息，然后通过 **FileIO** 类进行输入输出；
- 4) **jbox2d** 虽然可以自动模拟物理世界中组件的碰撞，但是其只有 **body** 刚体，不能很好地模拟弯轨的碰撞效果，该实验中只用简单方法改变其速度，并未很好模拟其贴臂下滑效果。

## 11. 参考

[JBox2D 物理引擎 \(dwenzhao.cn\)](http://dwenzhao.cn)

[jbox2d/jbox2d: a 2d Java physics engine, native java port of the C++ physics engines Box2D and LiquidFun \(github.com\)](https://github.com/jbox2d/jbox2d)

[yuchenECNU/GizmoBall: a easy ball game with the game engine -- Jbox2D \(github.com\)](https://github.com/yuchenECNU/GizmoBall)

