面向对象设计方法课程大作业实验报告

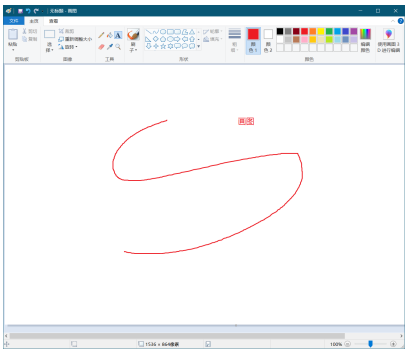
201220114 王思远

Saturday 21st May, 2022

第一部分　引言

第 1.1 节　开发任务简介

本次的开发任务是实现一个具有基础功能的图形绘制软件，需要拥有设计良好的图形用户界面，用户应该能够在软件种绘制一些基本的图形，并且能够添加图形描述。与此同时，我们的软件也应该能够支持对图形的拷贝与复制功能，并允许图形之间的组合复制与拷贝，同时也应该允许用户撤销上一步操作。显然，这是一个类似于 Windows 画板的软件。



Windows 内置的画图软件

第 1.2 节　采用的辅助技术框架

由于本次实验要求我们实现良好的图形界面，我们需要涉及计算机中的图形绘制问题，为了简化设计与缩短软件开发的周期，我选择使用额外的技术框架来实现这个软件。由于本人正在尝试学习 Unity 游戏引擎（也可以说是图形框架），因此决定尝试使用 Unity 框架来实现本次的设计。

如果你熟悉游戏或者游戏产业，那么想必你一定听说过 Unity 引擎了。Unity 引擎是一款跨平台游戏引擎，适用于全平台的软件开发，该引擎可以用于开发 2D 与 3D 游戏，交互式模拟以及其他类型的软件，很多非常流行的游戏，譬如《纪念碑谷》，《宝可梦 GO》，《茶杯头》，《逃离塔克夫》，包括国内许多耳熟能详的游戏，例如《明日方舟》，《鬼谷八荒》，《王者荣耀》甚至是《原神》都是使用该引擎开发的。除了游戏行业，该引擎还广泛应用在软件行业，尤其是可视化软件与可交互软件。

Unity 引擎

Unity 引擎

为什么选择 Unity 作为开发框架

Unity 引擎最大的优点就是简单易学的同时生产力强大，尤其适合开发人员少软件开发项目，并且由于此引擎最初是面向游戏开发的，因此非常适合图形化与可交互软件的开发。其使用的面向对象语言 C# 也是效率很高的面向对象语言。而对于软件工程来讲，最重要的就是开发效率和软件质量，因此我选择使用这个框架。

第 1.3 节　采用的面向对象语言

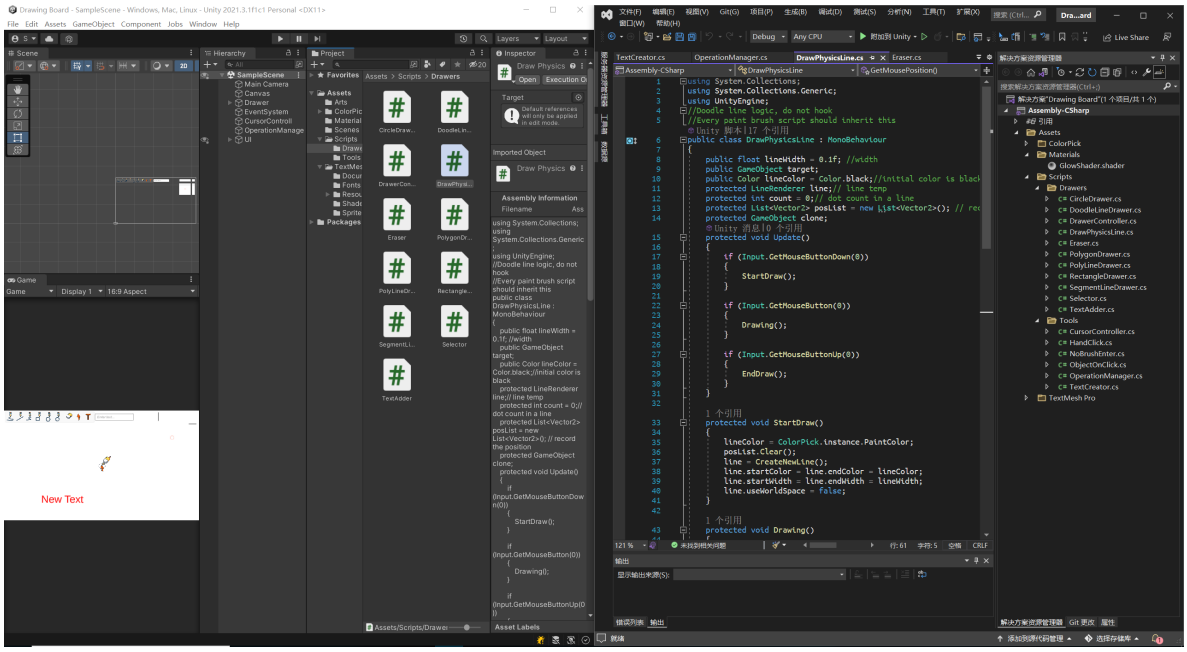
由于 Unity 框架使用 C# 语言作为其支持语言，因此本次开发使用的面向对象语言是 C#。这就不得不提到 C# 语言的特性以及 Unity 引擎中设计语言的组织方式了。

关于 C#

C# 是微软推出的一种基于.NET 框架的、面向对象的高级编程语言。是一种由 C 和 C++ 派生出来的面向对象设计语言。它在继承 C 和 C++ 强大功能的同时去掉了一些它们的复杂特性，使其成为 C 语言家族中的一种高效强大的编程语言。C# 以.NET 框架类库作为基础，使其具有全平台可移植的能力。在我看来，C# 就像一个 Java 与 C++ 的结合体，最吸引我的是其强大的快速开发能力，使用该语言，我能够在非常短的时间内实现复杂的功能，这要得益于其设计者对软件工程的深刻分析与面向对象设计方法的应用。

Unity 引擎中的 C# 语言

在 Unity 引擎中，代码是以脚本的形式存在的，我们所有的控制逻辑需要存储在不同的脚本中，而挂载好的脚本都会在引擎启动后被加载，我们通过脚本来对对象进行基本的控制。当然，我们使用脚本的方式和正常编写程序时的行为是基本相同的，只不过在编写自己的结构的同时，我们需要关注引擎与脚本对象交互的几个回调函数，这样我们才能借助引擎的力量实现自己的逻辑，提高开发的效率。



一个使用 C# 语言的 Unity 脚本

第二部分　对目标系统的分析与设计

第 2.1 节　进一步分析

在任务简介中，我已经初步分析了我们需要实现的功能，这里我们进一步分析一下每一种用户需求，来尝试获得一种设计思路。

有关图形绘制

根据要求，我们至少需要绘制默认大小的三角形，方框，圆形，椭圆，连接线等几种图形，这是作为一个图形绘制软件最基本的功能。观察这些图形，我们发现它们都是由线构成的，只要我们能够按照需求绘画出线段，我们就可以实现上述的所有功能，届时只需要对绘制线段的功能进行封装，用户就可以画出标准化的不同图形。因此，我们需要一种对象来表示线段，并使用线段绘制图形。

文字描述的添加

上述线段实现方法有一个无法实现的特例，就是文字，文字是不能用线段来拟合的，至少是不容易使用线段拟合的，因此，我们对文字应该使用另一种表示方式，由于从头开始绘制文字的工作量非常繁重，因此我们应该尽可能尝试使用框架的文字支持。

图形的选中，组合，复制与撤销

这一部分相对比较复杂，首先我们需要思考一些问题，如何选中一个图形？如何表示该图形被选中？用户怎么知道图形是否被选中？选中应该是图形对象的方法还是其他对象的方法？图形是否应该知道自己被选中？如何知道用户想要复制的是哪些图形？复制操作是谁的方法？撤销操作应该如何实现？怎么知道当前撤销到了哪一步？如何再次绘制撤销前的图形？这些问题非常繁杂，并且我们发现它们经常是密切相连的，如果我们使用相对分布式的设计方法解决这些问题，会大大提高程序设计的耦合性，降低程序的可拓展性，并且会提高漏洞的修复成本。因此，这里我们可以尝试使用中介者模式，使用一个中介者操控管理所有操作，降低程序的耦合度。

其他可能的需求

只要我们稍微使用一下只满足当前标准的软件，想必我们很快就会感觉到它具有很多的不足，需要添加一些新的功能，因此，我们的程序必须尽可能保持可拓展性，降低新增功能的成本，这就要求我们在设计时要尽量满足迪米特法则与开闭原则。

第 2.2 节　初步设计思路

在初步分析中，我们已经讨论了一些需求的可能实现方法，那么在这一步，我们将其转化为初步的设计思路：

图形

我们将每个图形都视为一个对象，这个对象是一个包含了一条线段的具体对象，对于不同的形状，我们只需更改线段中各个描点的位置，即可变换出不同的图形，因此我们对所有的图形应该是一视同仁的，都作为线段来看待，这将极大简化画笔的逻辑。

画笔

画笔是用来创建图形的，而由于图形具有一致性，我们可以在一个基础画笔的功能上实现新的画笔，因此我们可以使用继承来提高开发效率，我们先创建一个基类画笔，提供在屏幕上绘制图形的基本方法与逻辑，接下来的所有画笔只需继承这个基类画笔，并修改不同的逻辑，就可以实现了。与此同时，为了管理众多的画笔，我们需要一个管理者来控制画笔的生成与切换，为了符合开闭原则，我们需要对切换画笔的操作进行合理封装。

文字描述的添加

经过刚才的讨论，我们知道文字的图形拟合是比较复杂的，因此可以尝试使用另一种方法来添加文字。我们可以添加一个文字生成器，调用引擎的内部文字功能，生成对应的文字，我们只需要使用一个空画笔将其唤醒即可。

图形的选中，组合，复制与撤销

为了降低耦合度，这一步我们需要使用中介者模式，创建一个操作管理器对象作为中介，用户对于图形与操作的交互都要经过管理器来实现，这可以优化我们的逻辑拓扑，生成一个类似星形的结构。而为了满足迪米特原则，我们的图形对象对于所有对其自身的操作都应该是无知的，只有管理器知道应该如何操作。对于撤销与重做，我们可以通过备忘录模式来完成，记录由画笔生成并通知操作器添加，这样我们就把复杂的操作进行了一定程度的解耦。

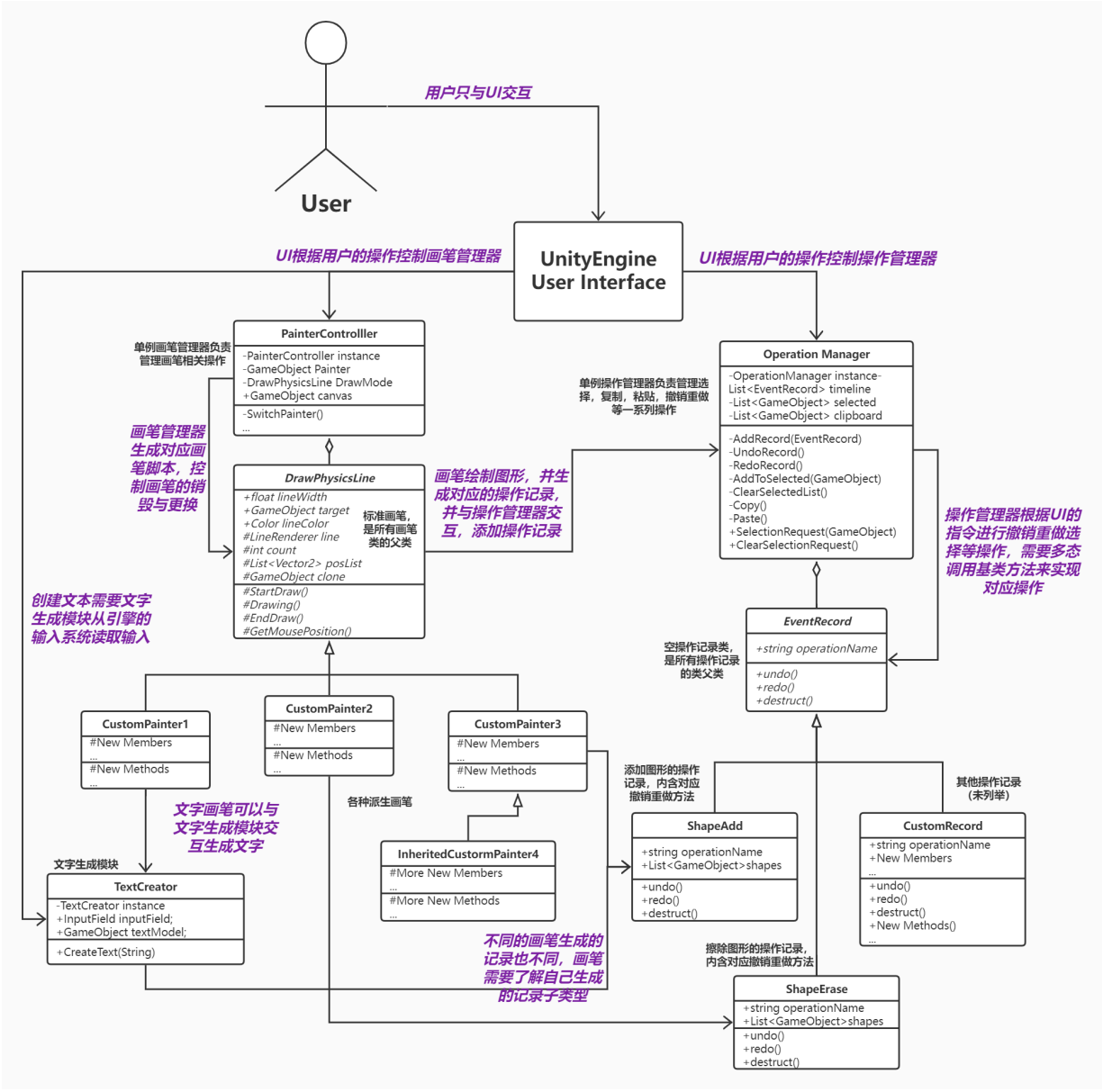
第 2.3 节　可用设计模式

在上面的分析中，我们已经提出了不少可用的模式，当然有许多模式是 Unity 引擎原生支持的，比如命令模式，观察者模式等等，这些就不纳入我的实现了，我们主要关注一下几种模式：

1. 单例模式（文字生成模块，操作管理器，画笔控制器）
2. 中介者模式（操作管理器）
3. 备忘录模式（操作管理器）
4. 桥接模式（画笔）
5. 状态模式（画笔）

第 2.4 节　UML 建模

在有预先逻辑的框架下，进行建模会有些复杂，不过我尽量避开框架中的逻辑，专注于程序独立于框架代码以外的逻辑，建模如下：



UML 类图

注意，这里没有给出包含在 Unity 引擎中的世界逻辑（相比之下那些逻辑没有上图列出的逻辑那么重要）!

第三部分　实现方案与设计模式详解

第 3.1 节　算法

本次的开发任务比较基础，并不需要特殊的算法，使用的算法仅包括排序，最大值最小值等算法，当然，我也使用这些基础算法设计了一些画笔的绘图算法，包括计算各个线段的点位置，长度，用户绘图中心等等，这里仅仅列举一下圆和椭圆画笔的一些计算对应线段的生成算法：

```
1  public class CircleDrawer : DrawPhysicsLine
2  {
3      public float R = 3;
4      public int N = 100;
5      Vector2 BasePoint;
6      Vector2 DestPoint;
7      Vector2 circleCenter;
8      // Update is called once per frame
9      new void Update()
10     {
11         if (Input.GetMouseButtonDown(0))
12             StartDraw();
13         if (Input.GetMouseButton(0))
14             Drawing();
15         if (Input.GetMouseButtonUp(0))
16             EndDraw();
17     }
18     new void StartDraw()
19     {
20         lineColor = ColorPick.instance.PaintColor;
21         BasePoint = GetMousePosition();//记录用户点击下的点位置，作为图形的基点
22         posList.Clear();
23         line = CreateNewLine();//创建一个新的线段对象
24         line.positionCount = N + 1;
25         line.startColor = line.endColor = lineColor;
26         line.startWidth = line.endWidth = lineWidth;
27         line.useWorldSpace = false;
28     }
29     new void Drawing()
30     {
31         float scaleX;//对X轴的缩放
32         float scaleY;//对Y轴的缩放
33         DestPoint = GetMousePosition();//用户鼠标当前所在的位置，应该能与图形基点构成一个矩形
34         circleCenter = (DestPoint + BasePoint) / 2;//计算矩形中心位置，作为圆的中心点
35         float xlength = Mathf.Abs(DestPoint.x - BasePoint.x);
36         float ylength = Mathf.Abs(DestPoint.y - BasePoint.y);
37         posList.Clear();
38         if (Input.GetKey("left shift"))//如果用户按下了shift键，那么用户会绘制出一个圆形
39         {
40             float squareWidth = Mathf.Max(Mathf.Abs(ylength), Mathf.Abs(xlength));
```

```
41            R = squareWidth / 2;//取当前鼠标位置与源点XY距离的最大值作为基准正方形边长
42            scaleX = 1;//圆的半径为基准正方形边长的一半，用户绘制出的圆形应该外接基准正方形
43            scaleY = 1;
44            for (int i = 0; i < N; i++)//根据三角函数与半径计算并设置线段点的位置
45            {
46                float x = R * Mathf.Cos((360f / N * (i + 1)) * Mathf.Deg2Rad) * scaleX + circleCenter.x;
47                float y = R * Mathf.Sin((360f / N * (i + 1)) * Mathf.Deg2Rad) * scaleY + circleCenter.y;
48                Vector2 point = new Vector2(x, y);
49                posList.Add(point);
50                line.SetPosition(i, point);
51            }
52            line.SetPosition(N, posList[0]);//为线段增加碰撞体
53        }
54        else
55        {
56            if (Mathf.Approximately(xlength, 0f) || Mathf.Approximately(ylength, 0f))
57                return;//若用户绘制x轴距离太短或者y轴距离太短，不能继续绘制，否则会导致除零错误
58            if (xlength > ylength)//x轴距离大于y轴距离时，椭圆是宽的
59            {//为了防止线段拟合圆形时露出裂口，开始绘制点应该在椭圆曲率最小的地方
60                scaleX = 1;
61                scaleY = ylength / xlength;//计算Y轴缩放率
62                R = xlength / 2;//计算半径
63                for (int i = 0; i < N; i++)//按照缩放绘制椭圆
64                {
65                    float x = R * Mathf.Cos((360f / N * (i + 1)) * Mathf.Deg2Rad) * scaleX + circleCenter.x;
66                    float y = R * Mathf.Sin((360f / N * (i + 1)) * Mathf.Deg2Rad) * scaleY + circleCenter.y;
67                    Vector2 point = new Vector2(x, y);
68                    posList.Add(point);
69                    line.SetPosition((i + (N / 4)) % N, point);
70                }
71                line.SetPosition(N, posList[(3 * N) / 4]);
72                //line.SetPosition(N + 1, posList[(3 * N) / 4 + 1]);
73            }
74            else//y轴距离大于x轴距离时，椭圆是扁的
75            {//为了防止线段拟合圆形时露出裂口，开始绘制点应该在椭圆曲率最小的地方
76                scaleY = 1;//除了开始绘制点不同，其余同理
77                scaleX = xlength / ylength;
78                R = ylength / 2;
79                for (int i = 0; i < N; i++)
80                {
81                    float x = R * Mathf.Cos((360f / N * (i + 1)) * Mathf.Deg2Rad) * scaleX + circleCenter.x;
82                    float y = R * Mathf.Sin((360f / N * (i + 1)) * Mathf.Deg2Rad) * scaleY + circleCenter.y;
83                    Vector2 point = new Vector2(x, y);
84                    posList.Add(point);
85                    line.SetPosition(i, point);
86                }
87                line.SetPosition(N, posList[0]);
88            }
89        }
90    }
91 }
```

第 3.2 节　数据结构

本次的开发任务比较基础，也不需要过于复杂的数据结构，全部数据结构均来自 C# 中自带的数据结构。当然，我使用了其中一些基本数据结构设计了一些功能稍微复杂一些的结构，这里举例撤销重做操作中使用到的时间线结构作为例子：

```
1  public class TimeLine//时间线是一个用于保存撤销与重做记录的结构，其实也是备忘录模式的一部分
2  {
3      private List<EventRecord> timeline = new List<EventRecord>();//使用C\#自带的List表来保存备忘录
4      private int current = 0;//current指针，永远指向当前，每一个新操作都会记录在current的位置
5      private int future = 0;//future指针，永远指向已经实现过的最远的未来操作
6      public void AddRecord(EventRecord record)//向时间线新增一条备忘录
7      {
8          while (future > current)//如果当前时间线有新增，那么未来就被永远的改变了
9          {
10             --future;
11             timeline[future].destruct();//原本的未来不会存在于当前的这条时间线中，可以直接销毁过时的未来备忘录
12             timeline.RemoveAt(future);
13         }
14         timeline.Add(record);//新增一条记录
15         ++current;//将current指针移动到下一个位置，准备继续在时间线上书写记录
16         future = current;//我们创造了一条新历史，因此当下就是最远的未来
17     }
18     private void UndoRecord()//撤销，相当于回到过去
19     {
20         if (current > 0)//我们最早也只能回到刚打开程序的时候
21         {
22             --current;//将current指针移到上一个位置，现在位于过去某点
23             EventRecord record = timeline[current]; //读取过去的某个操作备忘录，并调用其撤销方法
24             record.undo();
25         }
26         else
27         {
28             Debug.Log("Already at the earliest move...");
29         }
30     }
31     private void RedoRecord()//重做，相当于前进到已经发生过的未来
32     {
33         if (current < future)//只有未来曾经被创建过而且与当前的操作在同一时间线内，才有可能重做
34         {
35             EventRecord record = timeline[current];//读取未来某条操作备忘录，调用其重做方法
36             record.redo();
37             ++current;//将current指针移到下一个位置，现在位于未来某点
38         }
39         else
40         {
41             Debug.Log("Can not move further...");
42         }
43     }
44 }
```

第 3.3 节　设计模式

下面我们来讨论设计模式在本次实验中的应用，我们不妨沿着 2.3 节中给出的 5 种模式来讨论。

单例模式

单例模式是非常必要的经典模式，单例模式可以确保我们在任何情况下都不会生成一个类的两个实例，确保了实例的唯一性，提供一对一的受控访问，并且节约了系统的资源。单例模式在管理者的实现中是很有必要的，因为这确保了只有一个管理者能够对被管理的对象操作，否则非常容易出现一系列问题。

本次实验中，我在所有管理类中都应用了单例模式，比如画笔控制器与操作管理器，下面我终点分析一下单例模式在画笔控制器中的应用，在画笔控制器中，我们可以分离出如下代码：

```
public class PainterController : MonoBehaviour
{
    public static PainterController instance {get; private set;}
    // Start is called before the first frame Update
    void Start()
    {
        instance = this;
    }
    private PainterController(){}
}
```

这些代码看起来与我们课程中所讲的单例模式好像有些不一样，但是实际效果是相同的，C# 语言中基于访问控制提供了赋值取值器，如这里我们将 instance 实例的 get 属性设置为公有，而 set 属性设置为私有，就相当于对一个私有变量提供了一个公共设置方法。Unity 引擎在加载脚本的时候，会先生成一个该类对象，并调用其 start() 函数，因此这里我们只需要在这个函数内获得自身实例即可实现单例的初始化，显然这是一个饿汉式单例，我们也不需要考虑多线程问题。

中介者模式

中介者模式的最大作用是降低耦合度，我们可以稍加分析，如果图形的选择与复制功能是用户与图形直接交互的，那么每个图形都需要对用户的操作进行反应，若只有一两个图形，应该还没有什么大问题，但一旦图形数目增多，我们的逻辑就会非常混乱，用户与图形间的操作关系将会非常复杂，并且会非常消耗性能。因此我们引入了操作管理器作为中介者辅助我们管理图形，这样用户与图形之间就不需要显式的引用，进而降低了耦合度与复杂度。在操作管理器中，我们可以分离出如下代码：

```
public class OperationManager : MonoBehaviour
{
    private List<GameObject> selected = new List<GameObject>();
    private List<GameObject> clipBoard = new List<GameObject>();
    public void SelectionRequest(GameObject selectee);
    public void ClearSelectionRequest();
    private void Copy();
    private void Paste();
}
```

虽然很奇怪，但是我们的用户与图形之间确实存在某种同事关系，用户需要选中图形，图形需要根据用户需求删除或者复制自己。由于图形数量众多，让每个图形都同时与用户交互显然是不合理

的，这里我们就定义一个名为 selected 的同事对象表，用于存储图形的引用。当一个图形被用户点击后，就向中介者操作管理器发送一个选中请求，将自己添加进入该同事列表中，将自己与用户交互的权利移交给管理器，在用户进行操作的时候，中介人代为调用同事列表中图形对象的方法（比如销毁和拷贝自身等），这样就避免了用户与众多图形复杂的交互关系。

备忘录模式

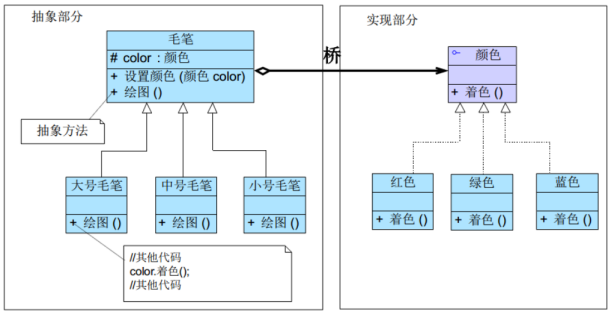
备忘录模式是实现撤销操作的不二之选，实现备忘录模式的代码我已经在 3.2 中有所描述了。这里我仅描述一下另一个有关数据结构，也就是备忘录对象本身。我们提取备忘录基类的代码，可以得到如下框架：

```
public class EventRecord
{
    public string operationName = "null";
    protected EventRecord() { }
    public EventRecord(string name)
    {
        operationName = name;
    }
    virtual public void undo();
    virtual public void redo();
    virtual public void destruct();
}
```

可以看出，这是一个备忘录对象，只不过由于画板操作的复杂性与多态性，我通过记录状态实现恢复不是很可行（保存画板全部状态的代价太大了），因此我将备忘录从状态改为记录状态的变化，即添加或者减少图形等其他操作，定义了恢复状态的函数，同时也定义了重做函数，这样我们也能在备忘录框架上更加方便地实现重做功能。在我们的代码中，我们定义的 TimeLine 结构就是一个拓展后的负责人与源发器的结合。

桥接模式

我在画笔中使用的桥接模式正如课中所讲的例子一样，画笔的颜色与画笔类型是分离开的，两者不存在耦合关系，正如图中所示：



讲义中桥接模式的例子

在本次项目中，画笔在绘图时才会调用颜色着色，我们把颜色与画笔类型之间的静态继承关系转化为了动态组合关系。

状态模式

用户在使用画笔作画时，一般不可能一次画出想要的结果，因此我们的画笔对象应该是动态的，在用户对其进行操作的不同时间，处理的事情应该也是不同的。

举一个例子，用户在绘制折线时，在用户没有点击之前，移动画笔应该不会产生任何作用，然而当用户点击之后，画笔应该在当前的位置记录一个折线点，并且在这之后用户移动画笔，应该能看到一条之前折线点与当前鼠标位置的线，这就涉及了一个状态的改变。下面是项目中折线绘制画笔的一段代码，可以看到这里应用了状态模式的设计思想，我们将绘制状态分为就绪与绘制过程中两个状态，若当前处在绘制状态，则调用生成新节点的方法，否则调用开始绘制方法。

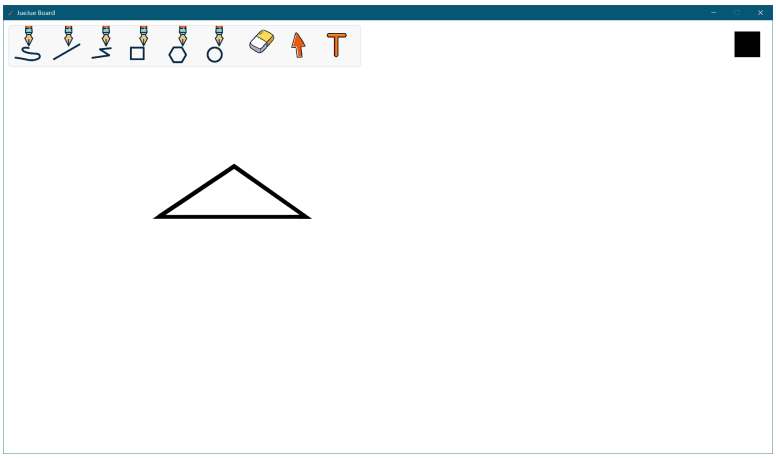
```
public class PolyLineDrawer : DrawPhysicsLine
{
    protected enum DrawingStatus {READY,INPROGRESS};
    protected DrawingStatus status = DrawingStatus.READY;
    new void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            if (status == DrawingStatus.READY)
            {
                StartDraw();
                status = DrawingStatus.INPROGRESS;
            }
            else if (status == DrawingStatus.INPROGRESS)
            {
                NewNode();
            }
        }
        if (status==DrawingStatus.INPROGRESS)
        {
            Drawing();
        }

        if (Input.GetMouseButtonDown(1) && status == DrawingStatus.INPROGRESS)
        {
            ......
            status = DrawingStatus.READY;
        }
    }
}
```

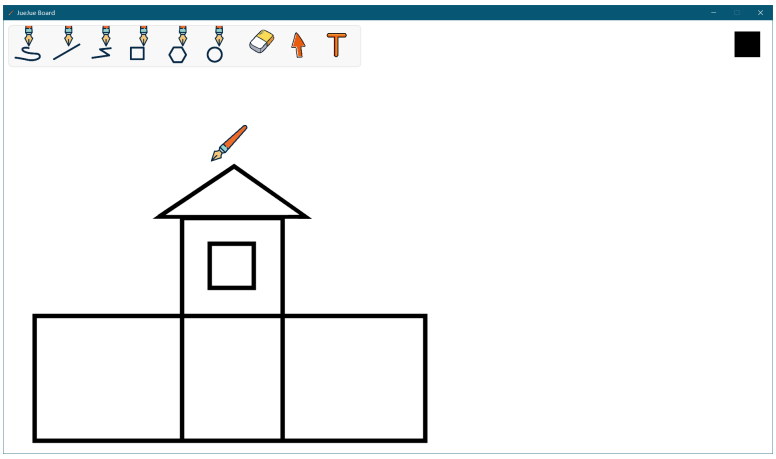
第四部分　实现功能介绍与效果展示

本次实验，我实现了所有要求的基础功能，并实现了部分拓展功能。

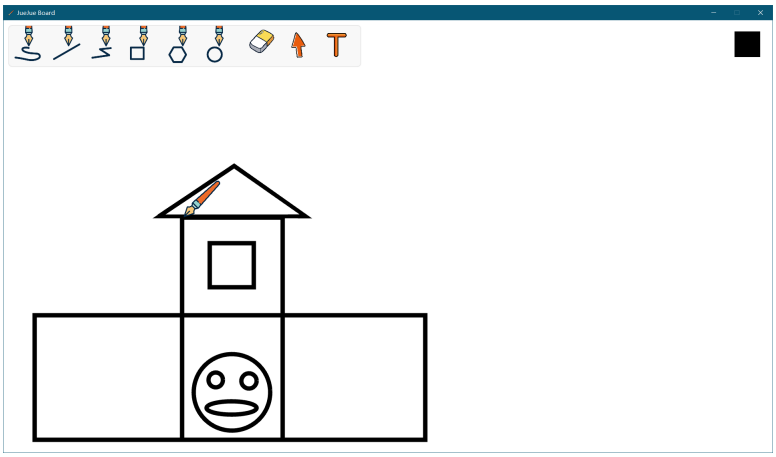
第 4.1 节　基础功能展示



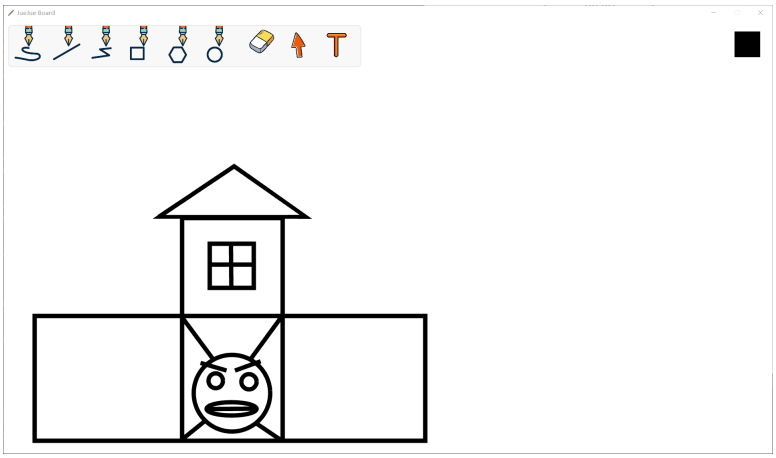
使用多边形工具即可绘制三角形



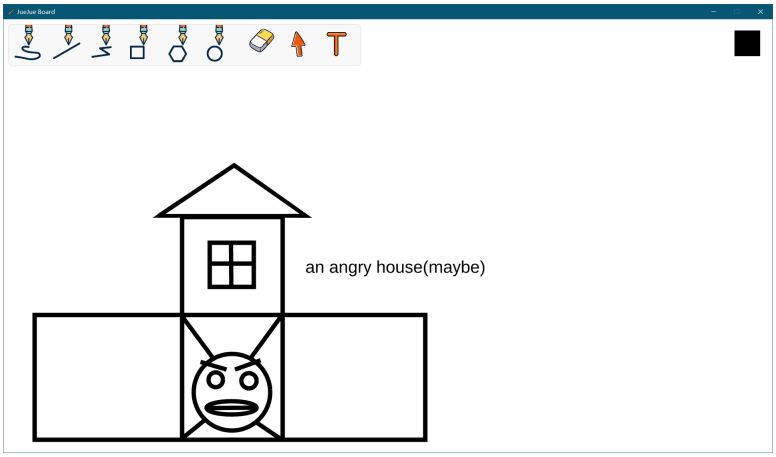
使用矩形工具即可绘制矩形，按住 shift 键可以绘制正方形



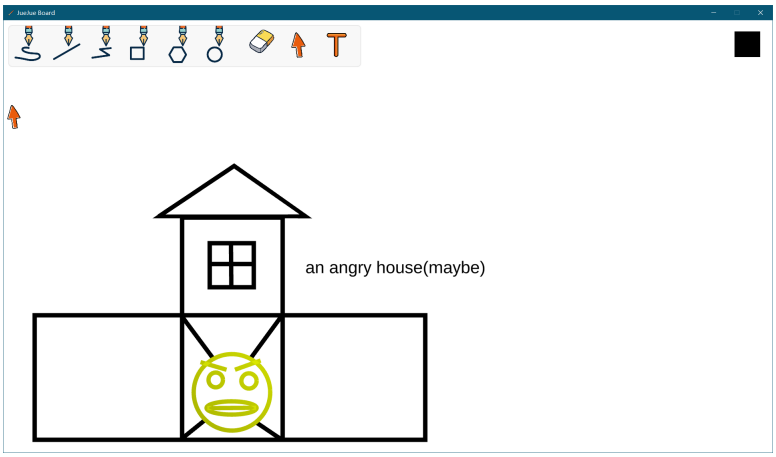
使用椭圆工具即可绘制椭圆，按住 shift 键可以绘制圆形



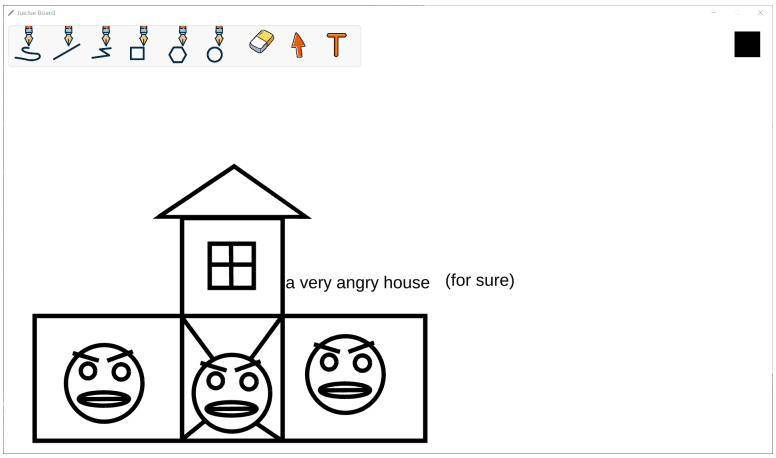
使用使用直线工具可以绘制连接线



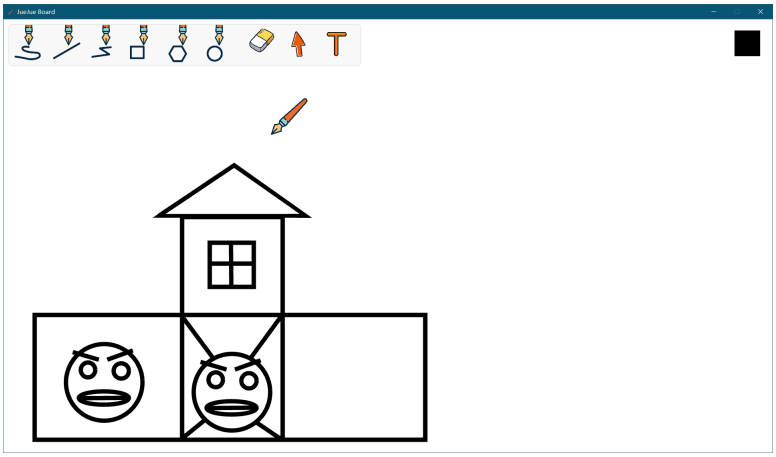
使用文字工具可以添加文字描述



使用选择工具选中一个或多个图形，被选中的图形会高亮闪烁，右键可以取消选中

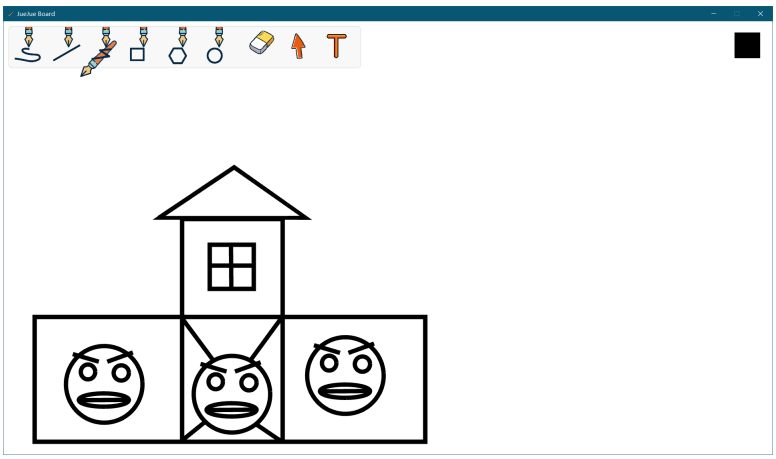


使用 Ctrl+C 对当前选中区域进行复制，使用 Ctrl+V 进行粘贴

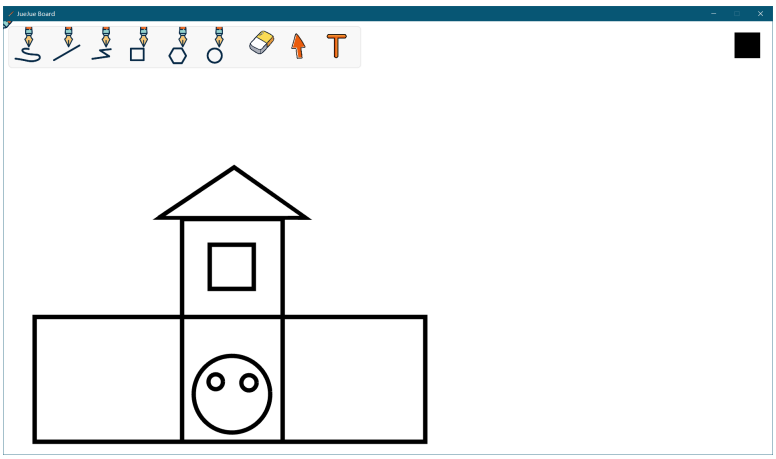


使用 Ctrl+Z 撤销上一步

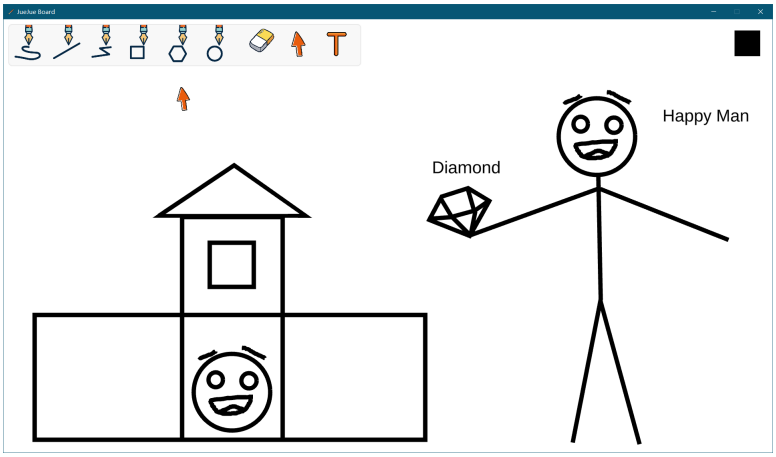
第 4.2 节　拓展功能展示



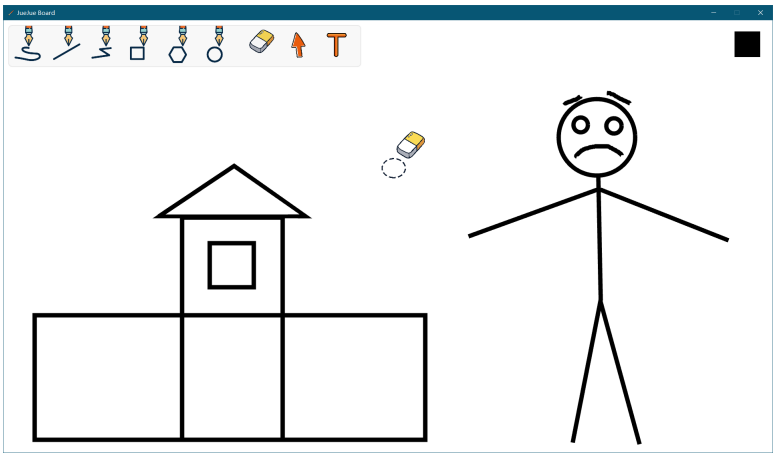
支持重做（Redo），使用 Ctrl+Y 来重做所有可能的操作



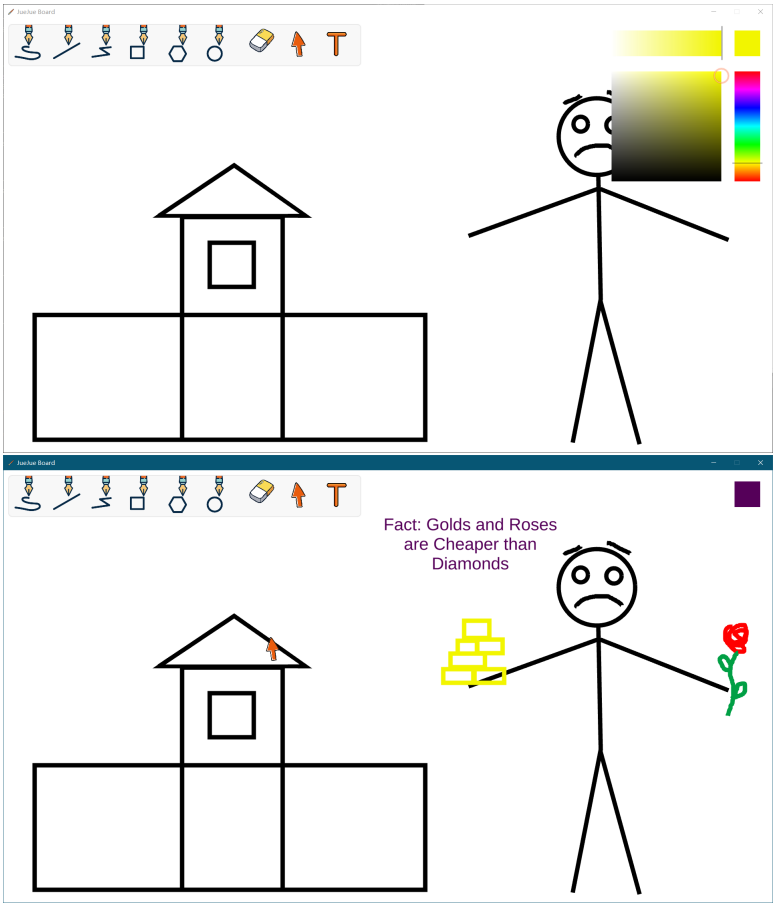
支持任意步撤销与重做，可在一个时间线内来回穿梭



新增涂鸦画笔，折线画笔与多边形画笔



新增橡皮擦，可擦除图形，擦除也可以撤销或重做



新增调色板，可以绘制出任意颜色甚至半透明的线

第五部分　小结

第 5.1 节　感想

本次实验花费了我较长的一段时间，由于课业任务仍然很重，我很少有一整天的时间来开发软件，因此我的工作都是断断续续的，然而正是这种间断的开发过程，让我深刻意识到了使用良好设计模式与面向对象设计方法的优点。在开发过程中我也遇到了不少难题，但是都能凭借思考来解决，这次实验相当程度上锻炼了我的开发能力，尽管我的代码仍有许多不足，开发出的软件可能也含有很多 Bug，但是无疑，经过这次实验，我对面向对象设计方法的理解提高了很多。

感谢阅读！