本科生课程实验报告

题　　目　电子信息系统综合设计课程实验报告

学　　院　电子信息学院

专　　业　通信工程

学生姓名　顾笑铭

学　　号　**2020141450003**　年级　**2020**

指导教师　李小舜

教务处制表

二〇二三年十二月六日

目录

第一章 摘要

本报告相关的试验工程完成于 2023 年 11 月 29 日，为四川大学电子信息学院 2020 级通信工程专业本科生顾笑铭与组员，对于必修课程“电子信息系统综合设计”完成的课程设计作业，一款针对跨平台的 GUI（Graphics User Interface）项目。

在本报告中仅涉及作者顾笑铭独立完成的工作，即基于 GLUT 的控件和俄罗斯方块游戏实现。报告将配合源代码展示该工程的具体实现。

第二章 GUI 控件

在这一章节，对按钮、开关、进度条和滑块四个控件的实现进行说明。

2.1 按钮

按钮的创建函数伴随的参数包括：预期的矩形按钮坐标、按钮上标记的字符串和一个函数指针即按钮对应的功能。

在头文件中，我们定义了一个枚举变量用以在鼠标未接触（NORMAl）、悬停（HOVER）和按下（PRESSED）三个不同的状态间切换，转换按钮的状态。

以下是按钮的头文件，“Button.h”。

```
// Button.h
#ifndef BUTTON_H
#define BUTTON_H

enum ButtonState { NORMAL, HOVER, PRESSED };

class Button
{
public:
    float x1, y1, x2, y2;
    const char* label;
    void(*action)();

    ButtonState state; // 将state作为类的成员变量

    float color[3];        // 按钮的默认颜色
    float hoverColor[3];   // 鼠标悬停时的颜色
    float pressedColor[3]; // 鼠标按下时的颜色

    // 构造函数
    Button(float x1, float y1, float x2, float y2, const char* l, void(*a)())
        : x1(x1), y1(y1), x2(x2), y2(y2), label(l), action(a), state(NORMAL) //
            在这里初始化state
    {
        color[0] = 0.9f; // 默认颜色
        color[1] = 0.9f;
        color[2] = 0.9f;

        hoverColor[0] = 0.7f; // 鼠标悬停时的颜色
        hoverColor[1] = 0.7f;
        hoverColor[2] = 0.7f;

        pressedColor[0] = 0.5f; // 鼠标按下时的颜色
        pressedColor[1] = 0.5f;
```

```
        pressedColor[2] = 0.5f;
    }

    bool isInside(float x, float y) const // 检查点是否在按钮内
    {
        return x >= x1 && x <= x2 && y >= y1 && y <= y2;
    }

    void drawButton() const;
};

#endif
```

以下是按钮的 cpp 文件，“Button.cpp”。

```
// Button.cpp
#include "Button.h"
#include <gl\GLUT.h>
#include <cstring>

void drawText(float x, float y, const char* text, void* font =
    GLUT_BITMAP_HELVETICA_12)
{
    glColor3f(0.0f, 0.0f, 0.0f);
    glRasterPos2f(x, y);
    while (*text)
    {
        glutBitmapCharacter(font, *text);
        text++;
    }
}

void Button::drawButton() const
{
    switch (state)
    {
    case HOVER:
        glColor3f(hoverColor[0], hoverColor[1], hoverColor[2]);
        break;
    case PRESSED:
        glColor3f(pressedColor[0], pressedColor[1], pressedColor[2]);
        break;
    default:
        glColor3f(color[0], color[1], color[2]);
    }

    glRectf(x1, y1, x2, y2);
    float textX = (x1 + x2) / 2 - 3 * strlen(label);
    float textY = (y1 + y2) / 2 + 3;
    drawText(textX, textY, label);
```

```
}
```

2.2 开关

开关与按钮的设计十分类似，区别在于需要在每次鼠标悬停未按下离开开关时，恢复至原状态，故增加一状态变量（flag），对应在主函数渲染开关图像时需要进行相应的调整。

以下是开关的头文件，“SwitchButton.h”。

```
// SwitchButton.h
#ifndef SWITCHBUTTON_H
#define SWITCHBUTTON_H

enum SwitchButtonState { OFF, HOLD, ON };

class SwitchButton
{
public:
    float x1, y1, x2, y2;
    const char* label;
    void(*action)();

    SwitchButtonState state; // 将state作为类的成员变量
    int flag = -1; // 开关状态

    float offColor[3];       // 开关断开时的颜色
    float holdColor[3];  // 鼠标悬停时的颜色
    float onColor[3];// 开关闭合时的颜色

    // 构造函数
    SwitchButton(float x1, float y1, float x2, float y2, const char* l, void(*a
        )())
    : x1(x1), y1(y1), x2(x2), y2(y2), label(l), action(a), state(OFF) // 在这里
        初始化state
    {
        offColor[0] = 0.9f; // 开关断开时的颜色
        offColor[1] = 0.9f;
        offColor[2] = 0.9f;

        holdColor[0] = 0.7f; // 鼠标悬停时的颜色
        holdColor[1] = 0.7f;
        holdColor[2] = 0.7f;

        onColor[0] = 0.5f; // 开关闭合时的颜色
        onColor[1] = 0.5f;
        onColor[2] = 0.5f;
    }

    bool isInsideSwitchButton(float x, float y) const // 检查点是否在开关内
    {
```

```
40          return x >= x1 && x <= x2 && y >= y1 && y <= y2;
41      }
42
43      void drawSwitchButton() const;
44  };
45
46  #endif
```

以下是开关的 cpp 文件，“SwitchButton.cpp”。

```
1   // SwitchButton.cpp
2   #include "SwitchButton.h"
3   #include <gl\GLUT.h>
4   #include <cstring>
5
6   void drawSwitchText(float x, float y, const char* text, void* font =
        GLUT_BITMAP_HELVETICA_12)
7   {
8       glColor3f(0.0f, 0.0f, 0.0f); // 文本颜色
9       glRasterPos2f(x, y);
10      while (*text)
11      {
12          glutBitmapCharacter(font, *text);
13          text++;
14      }
15  }
16
17  void SwitchButton::drawSwitchButton() const
18  {
19      switch (state)
20      {
21      case HOLD:
22          glColor3f(holdColor[0], holdColor[1], holdColor[2]);
23          break;
24      case ON:
25          glColor3f(onColor[0], onColor[1], onColor[2]);
26          break;
27      default:
28          glColor3f(offColor[0], offColor[1], offColor[2]);
29      }
30
31      glRectf(x1, y1, x2, y2);
32      float textX = (x1 + x2) / 2 - 3 * strlen(label);
33      float textY = (y1 + y2) / 2 + 3;
34      drawSwitchText(textX, textY, label);
35  }
```

2.3 进度条

进度条的创建函数伴随的参数包括：预期的矩形进度条坐标（宽和高）、进度条参量的最小值和最大值。

进度条不是一个直接与用户产生交互的控件，他与主程序中变量的变化密切相关。其构成即两个矩形的叠放展示，前景矩形随变量比例数值变化。

以下是进度条的头文件，“ProgressBar.h”。

```
//ProgressBar.h
#ifndef PROGRESSBAR_H
#define PROGRESSBAR_H

class ProgressBar
{
public:
    ProgressBar(float x, float y, float width, float height, float min, float
        max);

    void setValue(float value);
    void drawProgressBar() const;
    float max;

private:
    float x, y, width, height;
    float min, currentValue;
};

#endif
```

以下是进度条的 cpp 文件，“ProgressBar.cpp”。

```
//ProgressBar.cpp
#include "ProgressBar.h"
#include <gl/GLUT.h>

ProgressBar::ProgressBar(float x, float y, float width, float height, float min
    , float max)
    : x(x), y(y), width(width), height(height), min(min), max(max),
        currentValue(min) {}

void ProgressBar::setValue(float value)
{
    currentValue = value;
    if (currentValue < min) currentValue = min;
    if (currentValue > max) currentValue = max;
}

void ProgressBar::drawProgressBar() const
{
```

```
17      // 绘制背景
18      glColor3f(0.9f, 0.9f, 0.9f);
19      glRectf(x, y, x + width, y + height);
20
21      // 绘制前景（进度）
22      float progress = (currentValue - min) / (max - min);
23      glColor3f(0.0f, 1.0f, 0.0f);
24      glRectf(x, y, x + width * progress, y + height);
25  }
```

2.4 滑块

进度条的创建函数伴随的参数包括：预期的矩形进度条坐标（宽和高）、进度条参量的最小值和最大值。

以下是滑块的头文件，“Slider.h”。

```
1   // Slider.h
2   #ifndef SLIDER_H
3   #define SLIDER_H
4
5   class Slider
6   {
7   public:
8       float x1, y1, x2, y2; // 滑块的位置和尺寸
9       float minVal, maxVal; // 最小值和最大值
10      float currentValue;   // 当前值
11
12      Slider(float x1, float y1, float x2, float y2, float minVal, float maxVal)
13          : x1(x1), y1(y1), x2(x2), y2(y2), minVal(minVal), maxVal(maxVal),
                currentValue((maxVal + minVal)/2) {}
14
15      void setValue(float value); // 设置滑块的值
16      void drawSlider() const;    // 绘制滑块
17      bool isInside(float x, float y) const; // 检查点是否在滑块内
18      void moveSlider(float x, float y); // 移动滑块
19  };
20
21  #endif
```

以下是滑块的 cpp 文件，“Slider.cpp”。

```
1   // Slider.cpp
2   #include "Slider.h"
3   #include <gl/GLUT.h>
4
5   void Slider::setValue(float value)
6   {
7       currentValue = value;
8       if (currentValue < minVal) currentValue = minVal;
```

```
    if (currentValue > maxVal) currentValue = maxVal;
}

void Slider::drawSlider() const
{
    // 绘制滑块轨道
    glColor3f(0.5f, 0.5f, 0.5f);
    glRectf(x1, y1, x2, y2);

    // 计算滑块位置
    float sliderPosition = x1 + (currentValue - minVal) / (maxVal - minVal) * (
        x2 - x1);

    // 绘制滑块
    glColor3f(0.8f, 0.8f, 0.8f);
    glRectf(sliderPosition - 5, y1 - 10, sliderPosition + 5, y2 + 10);
}

bool Slider::isInside(float x, float y) const
{
    return x >= x1 && x <= x2 && y >= y1 - 10 && y <= y2 + 10;
}

void Slider::moveSlider(float x, float y)
{
    if (isInside(x, y))
    {
        float newValue = minVal + (x - x1) / (x2 - x1) * (maxVal - minVal);
        setValue(newValue);
    }
}
```

第三章 主函数

在这一章节，将展示使用上一章节控件的俄罗斯方块主函数。

3.1 函数库导入和全局参数定义

```
#include<gl/GLUT.H>
#include "Button.h"
#include "SwitchButton.h"
#include "ProgressBar.h"
#include "CheckBox.h"
#include "Slider.h"
#include <algorithm>

#define _USE_MATH_DEFINES
#include <math.h>

const int SIDE_PANEL_WIDTH = 200; // 显示框宽度

const int ROWS = 30; // 游戏版尺寸：30行
const int COLS = 10; // 游戏版尺寸：10列

int movedown_speed_original = 800; // 初始方块掉落速度：800ms触发一次
int movedown_speed = movedown_speed_original; // 方块掉落速度
int score = 0; // 游戏分数
int target = 200; // 游戏目标分数
int game_status = 1; // 游戏状态
int line_status = 1; // 框线状态
bool timerEnabled = true; // 计时器使能状态
```

3.2 定义方块类型和游戏版以及实例化

```
// 定义方块类型
enum BlockType
{
    I, O, T, S, Z, L, J, NUM_TYPES
};

const int BLOCK_SIZE = 4;
const int CELL_SIZE = 20; // 单元格像素数

// 定义方块形状
const bool SHAPES[NUM_TYPES][BLOCK_SIZE][BLOCK_SIZE] =
{
    // 0表示空，1表示存在单元格
    // 长条块 I
```

```
15      {
16          {0, 0, 0, 0},
17          {1, 1, 1, 1},
18          {0, 0, 0, 0},
19          {0, 0, 0, 0},
20      },
21
22      // 正方块 O
23      {
24          {0, 0, 0, 0},
25          {0, 1, 1, 0},
26          {0, 1, 1, 0},
27          {0, 0, 0, 0},
28      },
29
30      // 丁字块 T
31      {
32          {0, 0, 0, 0},
33          {0, 1, 1, 1},
34          {0, 0, 1, 0},
35          {0, 0, 0, 0},
36      },
37
38      // 右折叠块 S
39      {
40          {0, 0, 0, 0},
41          {0, 0, 1, 1},
42          {0, 1, 1, 0},
43          {0, 0, 0, 0},
44      },
45      // 左折叠块 Z
46      {
47          {0, 0, 0, 0},
48          {0, 1, 1, 0},
49          {0, 0, 1, 1},
50          {0, 0, 0, 0},
51      },
52      // 左转块 L
53      {
54          {0, 0, 0, 0},
55          {0, 1, 1, 1},
56          {0, 1, 0, 0},
57          {0, 0, 0, 0},
58      },
59      // 右转块 J
60      {
61          {0, 0, 0, 0},
62          {0, 1, 1, 1},
63          {0, 0, 0, 1},
64          {0, 0, 0, 0},
```

```
    }
};

// 定义方块状态
struct Block
{
    BlockType type;
    int x, y;  // 方块的左上角位置
    int rotation; // 0: 0°, 1: 90°, 2: 180°, 3: 270°
    bool shape[BLOCK_SIZE][BLOCK_SIZE];  // 当前的方块形状
};

// 定义游戏板
int board[ROWS][COLS] = { 0 };  // 0表示空, 1表示有方块

// 实例化当前方块
Block currentBlock;
```

3.3 游戏机制

```
// 定义操作: 旋转方块
void rotateBlock()
{
    bool temp[BLOCK_SIZE][BLOCK_SIZE];

    // 转置操作
    for (int i = 0; i < BLOCK_SIZE; i++)
    {
        for (int j = 0; j < BLOCK_SIZE; j++)
        {
            temp[j][i] = currentBlock.shape[i][j];
        }
    }

    // 反转列操作
    for (int i = 0; i < BLOCK_SIZE; i++)
    {
        for (int j = 0; j < BLOCK_SIZE / 2; j++)
        {
            bool tmp = temp[i][j];
            temp[i][j] = temp[i][BLOCK_SIZE - 1 - j];
            temp[i][BLOCK_SIZE - 1 - j] = tmp;
        }
    }

    // 更新当前方块的形状
    for (int i = 0; i < BLOCK_SIZE; i++)
    {
        for (int j = 0; j < BLOCK_SIZE; j++)
```

```
30          {
31              currentBlock.shape[i][j] = temp[i][j];
32          }
33      }
34  }
35
36  // 碰撞规则
37  bool checkCollision(int newX, int newY)
38  {
39      for (int i = 0; i < BLOCK_SIZE; i++)
40      {
41          for (int j = 0; j < BLOCK_SIZE; j++)
42          {
43              if (currentBlock.shape[i][j])
44              {
45                  if (newX + j < 0 || newX + j >= COLS || newY + i >= ROWS || (
                        newY + i >= 0 && board[newY + i][newX + j]))
46                  {
47                      return true;
48                  }
49              }
50          }
51      }
52      return false;
53  }
54
55  // 清除行规则
56  void checkAndClearRows()
57  {
58      for (int i = ROWS - 1; i >= 0; i--)
59      {
60          bool full = true;
61          for (int j = 0; j < COLS; j++)
62          {
63              if (!board[i][j])
64              {
65                  full = false;
66                  break;
67              }
68          }
69
70          if (full)
71          {
72              for (int k = i; k > 0; k--)
73              {
74                  for (int j = 0; j < COLS; j++)
75                  {
76                      board[k][j] = board[k - 1][j];
77                  }
78              }
```

```
79              for (int j = 0; j < COLS; j++)
80              {
81                  board[0][j] = 0;
82              }
83              i++;  // 因为整体下移了一行，所以需要再次检查当前行
84              score = score + 100;
85
86              if (score >= target)
87              {
88                  target = target * 5;
89                  score_progress.max = target;
90              }
91          }
92      }
93  }
94
95  // 产生新方块
96  void spawnBlock()
97  {
98      currentBlock.type = static_cast<BlockType>(rand() % NUM_TYPES);
99      currentBlock.x = COLS / 2 - BLOCK_SIZE / 2;  // 让方块出现在屏幕的中央位置
100     currentBlock.y = 0;
101     currentBlock.rotation = 0;  // 新方块初始旋转为0
102
103     for (int i = 0; i < BLOCK_SIZE; i++)
104     {
105         for (int j = 0; j < BLOCK_SIZE; j++)
106         {
107             currentBlock.shape[i][j] = SHAPES[currentBlock.type][i][j];
108         }
109     }
110
111     // 检查新方块是否立即产生碰撞，如果是，则游戏结束
112     if (checkCollision(currentBlock.x, currentBlock.y))
113     {
114         game_status = -1 * game_status;
115     }
116 }
117
118 // 放置方块
119 void placeBlock()
120 {
121     for (int i = 0; i < BLOCK_SIZE; i++)
122     {
123         for (int j = 0; j < BLOCK_SIZE; j++)
124         {
125             if (currentBlock.shape[i][j])
126             {
127                 // 如果方块的位置已经在最顶部，那么游戏结束
128                 if (currentBlock.y + i < 0) {
```

```
129                     game_status = -1; // 设置游戏状态为结束
130                     timerEnabled = false;
131                     return; // 直接返回，不放置方块
132                 }
133                 board[currentBlock.y + i][currentBlock.x + j] = 1;
134             }
135         }
136     }
137 
138     checkAndClearRows();
139     spawnBlock();
140 }
141 
142 // 方块下落
143 void moveDown()
144 {
145     // 尝试将方块向下移动一格
146     if (!checkCollision(currentBlock.x, currentBlock.y + 1))
147     {
148         currentBlock.y++;
149     }
150     else
151     {
152         // 如果无法向下移动，则放置方块并生成新方块
153         placeBlock();
154     }
155 
156     glutPostRedisplay();
157 }
158 
159 // 定时器
160 void tetris_timer(int value)
161 {
162     if (game_status == 1 && timerEnabled) // 游戏进行中且计时器启用
163     {
164         moveDown();
165     }
166     if (timerEnabled)
167     { // 这里再次检查以避免重复调用
168         glutTimerFunc(movedown_speed, tetris_timer, 0); // 重新设置计时器
169     }
170 }
171 
172 // 用户键盘操作
173 void tetris_keyboard(unsigned char key, int x, int y)
174 {
175     switch (key)
176     {
177     case 'a':
178         // 向左移动方块，前提是没有碰撞
```

```
179             if (!checkCollision(currentBlock.x - 1, currentBlock.y))
180             {
181                 currentBlock.x--;
182             }
183             break;
184 
185         case 'd':
186             // 向右移动方块
187             if (!checkCollision(currentBlock.x + 1, currentBlock.y))
188             {
189                 currentBlock.x++;
190             }
191             break;
192 
193         case 's':
194             // 加速方块下落
195             while (!checkCollision(currentBlock.x, currentBlock.y + 1))
196             {
197                 currentBlock.y++;
198             }
199             break;
200 
201         case 'w':
202             // 旋转方块
203             rotateBlock();
204             if (checkCollision(currentBlock.x, currentBlock.y))
205             {
206                 // 如果旋转后产生碰撞，则撤销旋转
207                 rotateBlock();
208                 rotateBlock();
209                 rotateBlock();
210             }
211             break;
212     }
213 
214     glutPostRedisplay();
215 }
```

3.4 UI 控件耦合

```
1 // 按钮、开关功能创建
2 void restartGame();
3 void toggleBorders();
4 
5 Button restartButton(COLS* CELL_SIZE + 20, 110, COLS* CELL_SIZE + 200, 140, "
      Restart", restartGame);
6 
7 SwitchButton toggleSW(COLS* CELL_SIZE + 20, 150, COLS* CELL_SIZE + 200, 180, "
      Toggle␣Borders", toggleBorders);
```

```
8
9  void restartGame()
10 {
11     score = 0; // 重置分数
12     game_status = 1; // 重置游戏状态
13
14     // 清空游戏板
15     for (int i = 0; i < ROWS; i++) {
16         for (int j = 0; j < COLS; j++) {
17             board[i][j] = 0;
18         }
19     }
20
21     timerEnabled = false; // 先禁用计时器
22     spawnBlock(); // 生成新的方块
23     timerEnabled = true; // 启用计时器
24 }
25
26 void toggleBorders()
27 {
28     line_status = -line_status;
29 }
30
31 // 复选框
32 // CheckBox checkBox(50, 80, 20);
33
34 // 滑块
35 Slider slider(COLS* CELL_SIZE + 20, 250, COLS* CELL_SIZE + 200, 260, 0.5, 5);
36
37 // 进度条
38 ProgressBar score_progress(COLS* CELL_SIZE + 20, 200, 180, 30, 0, target);
39
40 // 鼠标点击检测
41 void MouseClicked(int button, int state, int x, int y)
42 {
43     if (button == GLUT_LEFT_BUTTON)
44     {
45         if (state == GLUT_DOWN)
46         {
47             // 鼠标按下事件，更新按钮状态为 PRESSED
48             if (restartButton.isInside(x, y))
49             {
50                 restartButton.state = PRESSED;
51             }
52
53             // 鼠标按下事件，更新开关状态
54             if (toggleSW.isInsideSwitchButton(x, y))
55             {
56                 if (toggleSW.flag == 1)
57                 {
```

```
58                      toggleSW.state = OFF;
59                      toggleSW.flag = -toggleSW.flag;
60                  }
61                  else
62                  {
63                      toggleSW.state = ON;
64                      toggleSW.flag = -toggleSW.flag;
65                  }
66              }
67          }
68          else if (state == GLUT_UP)
69          {
70              // 鼠标释放事件，如果在按钮内则调用按钮动作
71              if (restartButton.isInside(x, y))
72              {
73                  restartButton.action(); // 调用按钮动作
74                  restartButton.state = NORMAL;
75              }
76
77              // 鼠标释放事件，如果在开关内则调用按钮动作
78              if (toggleSW.isInsideSwitchButton(x, y))
79              {
80                  toggleSW.action(); // 调用开关动作
81              }
82          }
83      }
84
85      slider.moveSlider(x, y);
86      movedown_speed = movedown_speed_original / slider.currentValue;
87
88      glutPostRedisplay();
89  }
90
91  // 鼠标移动检测
92  void MouseMoved(int x, int y)
93  {
94      restartButton.state = restartButton.isInside(x, y) ? HOVER : NORMAL;
95
96      if (toggleSW.flag == -1)
97      {
98          toggleSW.state = toggleSW.isInsideSwitchButton(x, y) ? HOLD : OFF;
99      }
100     else
101     {
102         toggleSW.state = toggleSW.isInsideSwitchButton(x, y) ? HOLD : ON;
103     }
104
105     glutPostRedisplay();
106 }
```

3.5 图窗展示

```
// 初始化图窗
void initOpenGL()
{
    glClearColor(0.0, 0.0, 0.0, 1.0);  // 设置背景色为黑色
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    int windowWidth = 10 + COLS * CELL_SIZE + SIDE_PANEL_WIDTH + 10;
    int windowHeight = 10 + ROWS * CELL_SIZE + 10;
    gluOrtho2D(0, windowWidth, windowHeight, 0); // 设置窗口的宽高
}

// 绘制元素格
void drawBlock(int x, int y)
{
    glRecti(x * CELL_SIZE, y * CELL_SIZE, (x + 1) * CELL_SIZE, (y + 1) *
        CELL_SIZE);
}

// 展示函数
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);

    // 绘制游戏板上的方块
    glColor3f(1.0, 1.0, 1.0);
    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            if (board[i][j])
            {
                drawBlock(j, i);
            }
        }
    }

    // 绘制当前方块
    glColor3f(1.0, 0.0, 0.0);
    for (int i = 0; i < BLOCK_SIZE; i++)
    {
        for (int j = 0; j < BLOCK_SIZE; j++)
        {
            if (currentBlock.shape[i][j])
            {
                drawBlock(currentBlock.x + j, currentBlock.y + i);
            }
        }
```

```
47      }
48
49      // 绘制分割线
50      if (line_status == 1)
51      {
52          for (int i = 0; i < ROWS; i++)
53          {
54              for (int j = 0; j < COLS; j++)
55              {
56                  if (board[i][j])
57                  {
58                      glColor3f(0.5, 0.5, 0.5);
59                      //glRectf(j * BLOCK_SIZE, i * BLOCK_SIZE, (j + 1) *
                            BLOCK_SIZE, (i + 1) * BLOCK_SIZE);
60                  }
61
62                  // 绘制分割线，颜色根据设计需要进行更改
63                  glColor3f(1.0, 1.0, 1.0);  // 白色线
64                  glBegin(GL_LINES);
65                  glVertex2f(j * CELL_SIZE, i * CELL_SIZE);
66                  glVertex2f(j * CELL_SIZE, (i + 1) * CELL_SIZE);
67                  glVertex2f(j * CELL_SIZE, i * CELL_SIZE);
68                  glVertex2f((j + 1) * CELL_SIZE, i * CELL_SIZE);
69                  glEnd();
70              }
71              // 保证每行的最右边有线，每行的末尾绘制一个垂直线段
72              glColor3f(1.0, 1.0, 1.0);
73              glBegin(GL_LINES);
74              glVertex2f(COLS * BLOCK_SIZE, i * BLOCK_SIZE);
75              glVertex2f(COLS * BLOCK_SIZE, (i + 1) * BLOCK_SIZE);
76              glEnd();
77          }
78      }
79
80      // 绘制右侧的分数面板
81      char scoreText[50];
82      sprintf_s(scoreText, sizeof(scoreText), "Score:␣%d", score);
83
84      char targetText[50];
85      sprintf_s(targetText, sizeof(targetText), "Target:␣%d", target);
86
87      glColor3f(1.0, 1.0, 1.0);
88      glRasterPos2i(COLS * CELL_SIZE + 10, 30); // 设置实时分数显示位置
89
90      for (char* c = scoreText; *c != '\0'; c++)
91      {
92          glutBitmapCharacter(GLUT_BITMAP_9_BY_15, *c);
93      }
94
95      glColor3f(1.0, 1.0, 1.0);
```

```
96      glRasterPos2i(COLS * CELL_SIZE + 10, 50); // 设置目标分数显示位置
97
98      for (char* c = targetText; *c != '\0'; c++)
99      {
100         glutBitmapCharacter(GLUT_BITMAP_9_BY_15, *c);
101     }
102
103     restartButton.drawButton();
104
105     toggleSW.drawSwitchButton();
106
107     // checkBox.drawBox();
108
109     score_progress.setValue(score);
110
111     score_progress.drawProgressBar();
112
113     slider.drawSlider();
114
115     glutSwapBuffers();
116 }
```

3.6 main 函数

```
1  int main(int argc, char** argv)
2  {
3      glutInit(&argc, argv);
4      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
5
6      int windowWidth = 10 + COLS * CELL_SIZE + SIDE_PANEL_WIDTH + CELL_SIZE;
7      int windowHeight = 10 + ROWS * CELL_SIZE + 10;
8      glutInitWindowSize(windowWidth, windowHeight);
9      glutCreateWindow("Tetris");
10     initOpenGL();
11
12     // 注册回调函数
13     glutTimerFunc(500, tetris_timer, 0);
14     glutDisplayFunc(display);
15     glutKeyboardFunc(tetris_keyboard);
16     glutPassiveMotionFunc(MouseMoved);
17     glutMouseFunc(MouseClicked);
18
19     spawnBlock();  // 生成初始方块
20     glutMainLoop();
21     return 0;
22 }
```

第四章 效果展示

代码公布于 https://github.com/NoKi41/Study-Document/tree/GUI_Tetris_CPP_GLUT