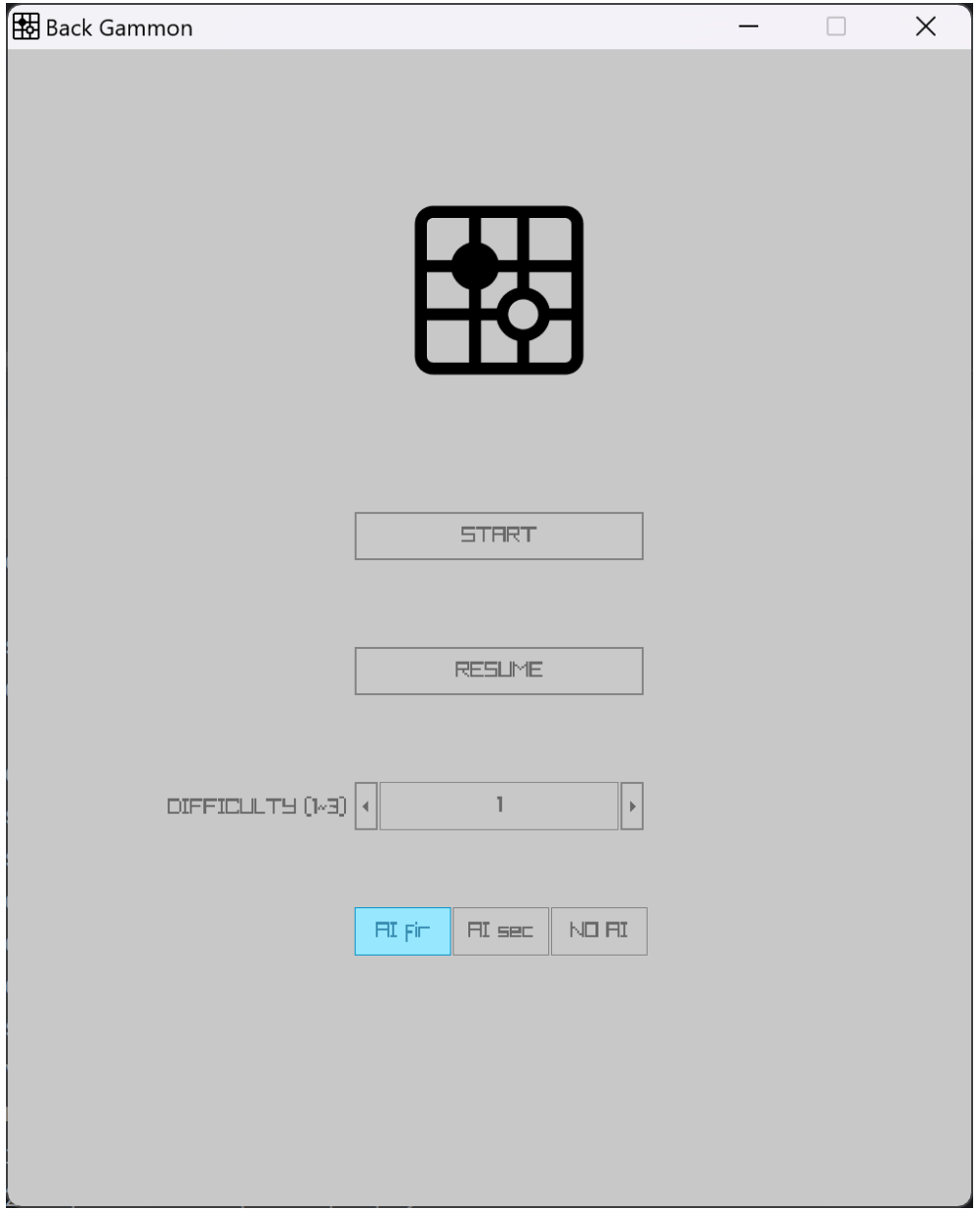
AI五子棋

需求

- 用户界面
- 游戏模式
- AI算法
- 数据存储
- 用户交互

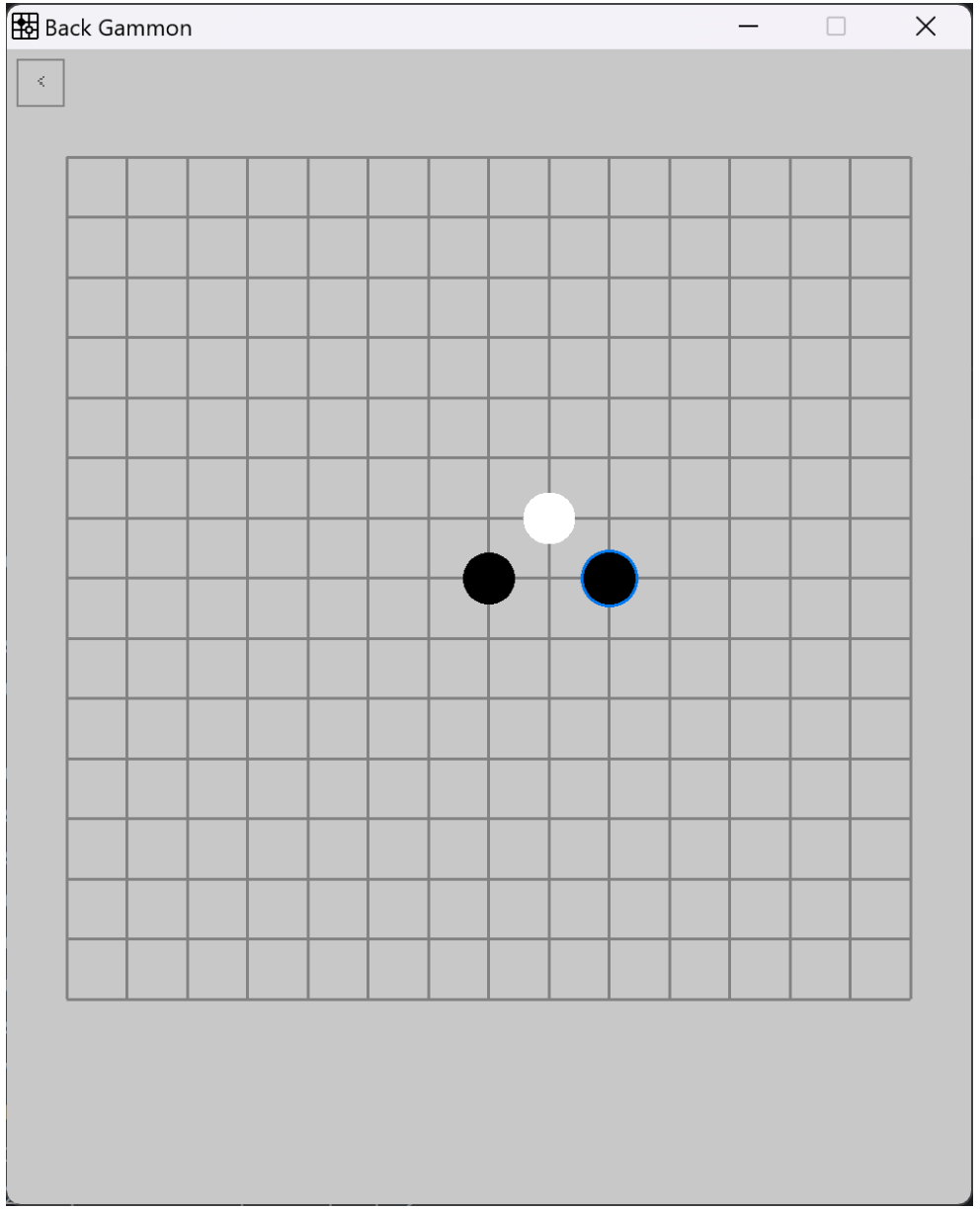
程序外观

- 主界面



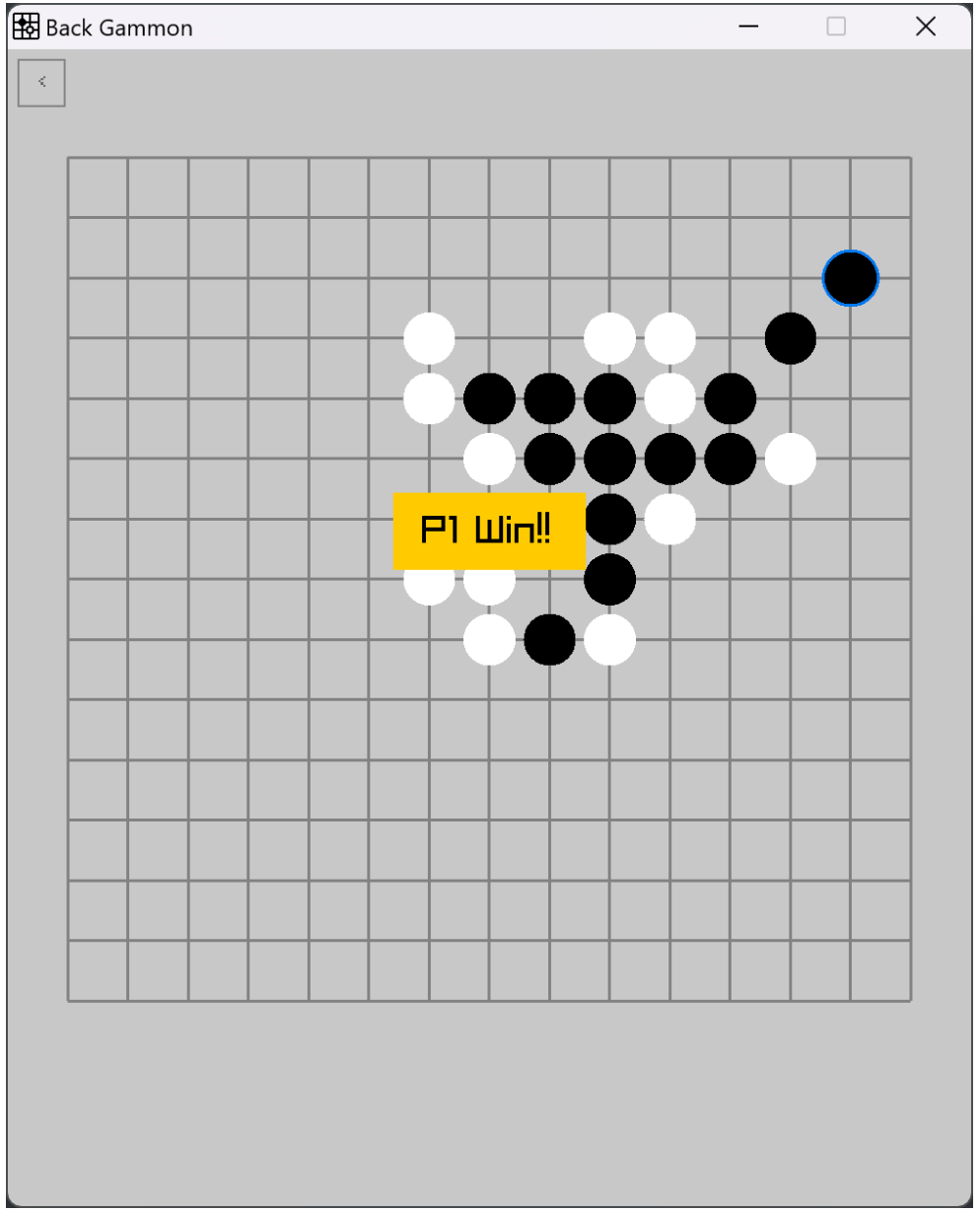
程序外观

- 游戏界面



程序外观

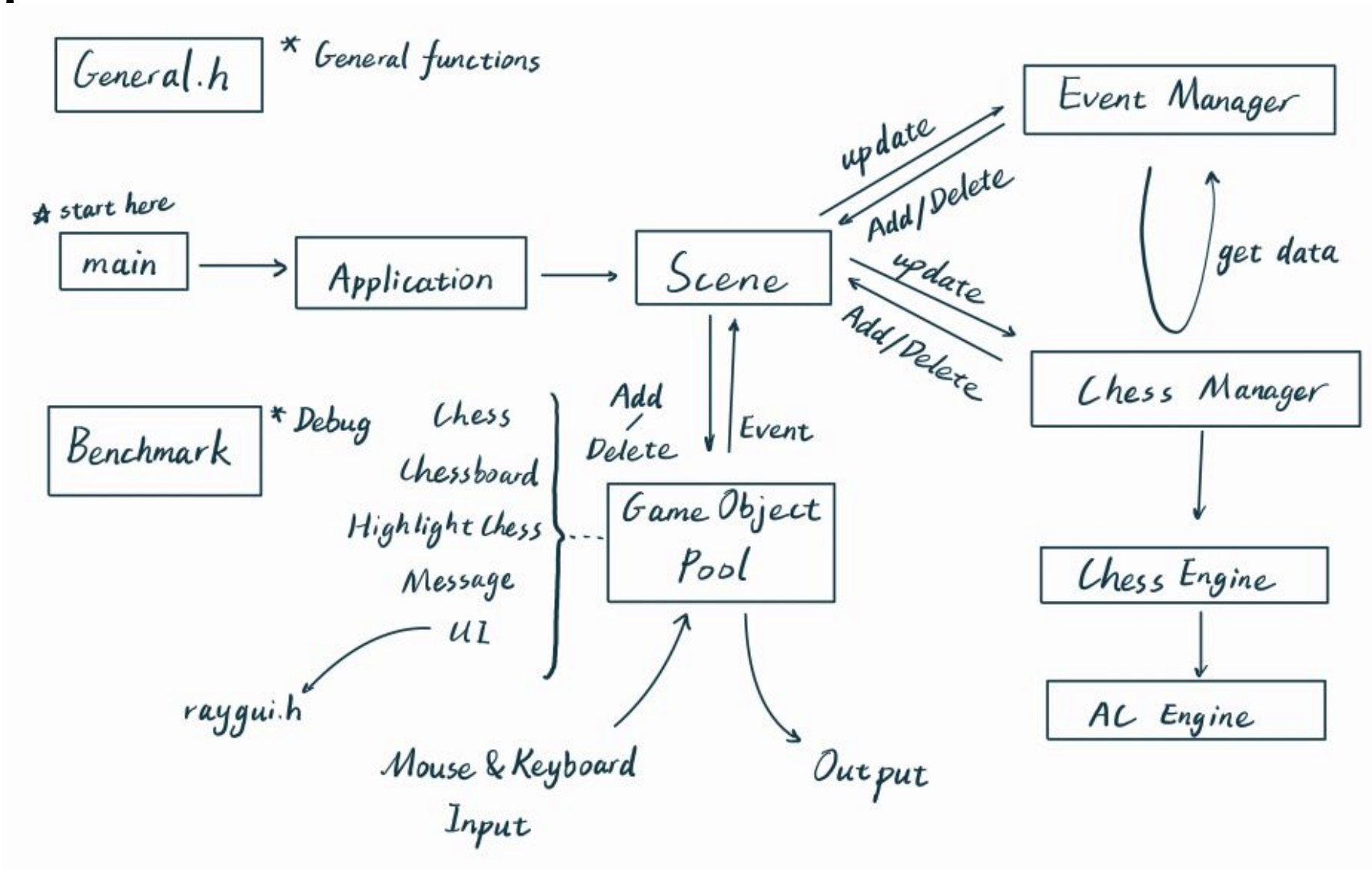
- 胜利界面



操作方式

- 鼠标点按（符合直觉的）
- 方向左键：撤回（隐藏功能 不提示用户）
- 方向右键：电脑帮忙下一个子（在人机对战的情况下）（隐藏功能 不告诉用户）

设计


General.h
* General functions
Event Manager
update
Add/Delete
start here
main
Application
Scene
update
Add/Delete
get data
Chess Manager
Benchmark
* Debug
Chess
Chessboard
Highlight Chess
Message
UI
Add
Delete
Event
Game Object Pool
Chess Engine
AC Engine
raygui.h
Mouse & Keyboard Input
Output

框架的搭建

- 最初的想法和现在差不多，考虑游戏围绕update & draw运行，不过event的解耦始终非常困难，因为其中的决策又要用到很多棋盘的信息，不能只靠一个event触发就无脑运行一个功能（如悔棋）
- 于是一开始设计成Scene决策和绘画一体的模式

框架的搭建

```
void Scene::Update() {
    if (!IsWindowsStatic) {//窗口能获取信息
        if (cnt % 2) {//到电脑下了
            ...
            return;
        }

        if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
            ...

            if (SomeoneWin(roundVec)) {
                m_gameObjects.push_back(new Message(cnt % 2 ? "P1 Win!!" : "P2 Win!!"));
                IsWindowsStatic = true;
            }
            return;
        }

    }
}
```

框架的搭建

- 很明显非常混乱
- 而且不利于debug
- 所以进行包装，用函数管理

```
void Scene::Update() {
    if (IsWindowsStatic) return;
    if (isWantToCancel()) {
        cancelOneStep();
        if (haveComputerHere()) {
            cancelOneStep();
        }
    }
    if (isWantToSwitchPlayer()) {
        switchPlayer();
    }
    if (isComputerNow()) {
        computerDownOneStep();
    }
    if (isHunanNow() and isPlayerClickOnBoardAndValid()) {
        humanDownOneStep();
    }
    if (someoneWin()) {
        IsWindowsStatic = true;
        showWinMessage();
    }
}
```

框架的搭建

- 这个在单单棋盘界面的时候是没有问题的，但是一旦加进来主菜单的时候就会非常混乱了
- 所以又把event请回来了，同时为了方便把event Manager、chess Manager和Scene都弄成单例的模式
- 后来发现多此一举，又全部弄成静态的方法来管理
- 最后就变成了这样的：

框架的搭建

- 能够画出来的对象都有统一的接口，所以绘画非常方便
- 之后update就交给Event Manager来决策了，那么信息如何传递呢，我用的是Enum枚举

```
void Scene::Update() {
    for (GameObject *obj: m_gameObjects) {
        if (obj->m_isActive) {
            EventManager::AddEvent(obj->Update());
        }
    }
    if (EventManager::IsGaming()) {
        EventManager::AddEvent(ChessManager::update());
    }
    EventManager::Update();
}
```

框架的搭建

- 任何一个出现在游戏里的对象都能够通过update返回一个event来激活一个事件。而这个event最终会传递event Manager处理。Event Manager把event统一暂存（优先队列）再统一处理。

```
enum Event {
    EVENT_NONE = 0,
    EVENT_IS_WANT_TO_RETURN_TO_MAIN_MENU = 1,

    ...

    EVENT_SET_GAME_MODE_AI_SECOND = 210,
    EVENT_SET_GAME_MODE_NO_AI = 211,
};
```

框架的搭建

- 任何一个出现在游戏里的对象都能够通过update返回一个event来激活一个事件。而这个event最终会传递event Manager处理。Event Manager把event统一暂存（优先队列）再统一处理。
- Event Manager的处理：
- 这样便很好的分离了决策和绘画的功能，并且加强了程序的扩展性。

```
void EventManager::Update() {
    while (!m_eventQueue.empty()) {
        Event event = m_eventQueue.top();
        m_eventQueue.pop();
        switch (event) {
            case EVENT_IS_WANT_TO_CANCEL:
                cancelOneStep();
                if (!ChessManager::thereIsNoComputer()) {
                    cancelOneStep();//人机对战模式撤回两次
                }
                break;

            ...

            case EVENT_IS_WANT_TO_RETURN_TO_MAIN_MENU:
                ChessEngine::searchFloor = 3;
                ChessManager::computerIsPWhat = 1;
                IS_GAMING = false;
                IS_GAME_OVER = false;
                Scene::m_gameObjects.clear();
                Scene::AddObject(new UI);
                ChessManager::saveState();
                ChessManager::clear();
                break;
            case EVENT_NONE:
                break;
        }
    }
}
```

核心算法模块

- 一开始没有这个模块而是把它和chess Manager放在一起，但是考虑到分离管理和计算两个任务，所以分了这个模块出来

- Chess Engine对外的接口有4个：

- Init Map 用于获取棋盘的状态数组。元素必须为0，1，2中的一个
- Get Max Coord 用于获取当前棋盘分数最高的坐标位置（也就是最优解的坐标）
- someone Win用来确认当前是否有人已经赢了
- Search Floor储存搜索层数，用来调节难度

核心算法改进过程

- 一开始我使用最大值搜索
- 用排序来实现
- 虽然很简洁明了，但是棋力不够

```
iVector2 AI::DownCoord() {
    std::vector<DownStruct> possibleDown = getPossibleDown();

    for (auto nowDown: possibleDown) {
        nowDown.calcDownScore();
    }

    std::sort(possibleDown.begin(), possibleDown.end());

    return possibleDown[0].coord;
}
```

核心算法改进过程

- 然后采用ab剪枝，虽然可以搜到4层了，但是非常慢，剪枝几乎没有效果

```
Leaf AI::AlphaBeta(int depth, Leaf alpha, Leaf beta, iVector2 nowDown) {
    Leaf ret{};
    if (depth == 0) {        //到达一定深度，评估棋局，返回分值
        return {nowDown, score_sum};
    }
    std::queue<iVector2> possibleStep = generalPossibleStep();
    while (!possibleStep.empty()) {
        //下棋
        nowDown = possibleStep.front();
        possibleStep.pop();
        DownUpdateScore(nowDown);
        ret = AlphaBeta(depth - 1, alpha, beta, nowDown);
        //撤销
        UpUpdateScore(nowDown);
        //剪枝
        if (depth % 2 == 0) {//MAX层
            if (ret > alpha) {
                alpha = {nowDown, ret.score};
            }
            if (alpha > beta) {
                return {nowDown, INT_MAX};//返回无效贡献值
            }
        } else {//MIN层
            if (beta > ret) {
                beta = {nowDown, ret.score};
            }
            if (alpha > beta) {
                return {nowDown, INT_MIN};
            }
        }
    }
    return depth % 2 ? beta : alpha;
}
```

核心算法改进过程

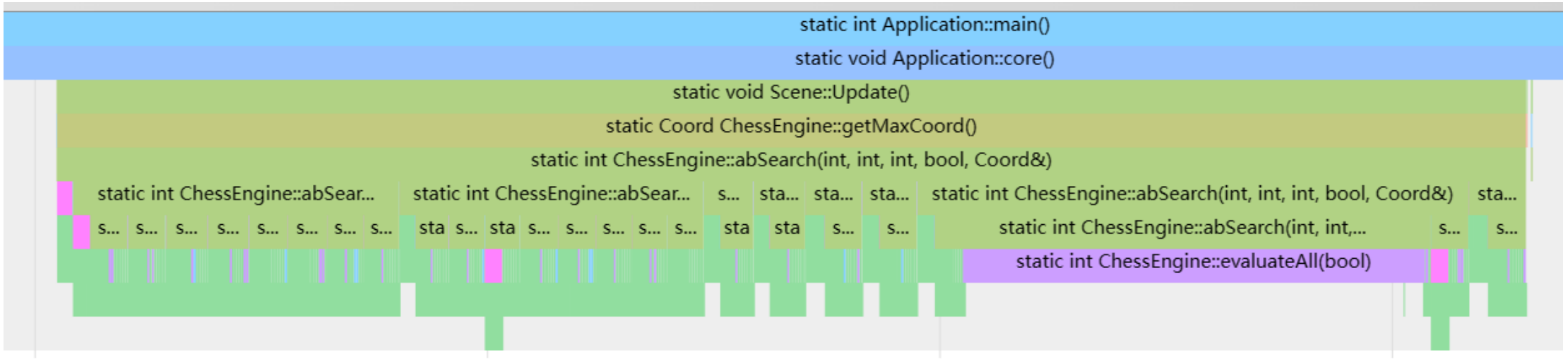
- 这个时候我因为对scene也不够满意，所以直接把AI部分重写了一遍，然后把运算部分全部打包在chess engine里，再重写的过程中偶然发现了为什么剪枝效果非常差的原因：是因为在对一个点进行评估的是由没有考虑两边，而是单单考虑到了要下的子在这个点的分数。
- 于是考虑分数做差，这步棋由双方各下一次然后分数相减，差值越大说明越要占该点。
- 加上这个改进之后剪枝的效果大增，然后可以只考虑前面的几种可能（分数最大的）来减少搜索的事件，变相增加搜索的深度。
- 这样棋力又进一步增加，搜索深度可以到6层，单步小于10秒思考。

生成器

- 最后洗牌算法
- 让结果随机化

```
std::vector<ScoreCoord> ChessEngine::generatePossibleMove(bool isBlackNow) {
    PROFILE_FUNCTION
    std::vector<ScoreCoord> ret;
    ret.reserve(225);
    for (int x = 1; x <= 15; ++x) {
        for (int y = 1; y <= 15; ++y) {
            if (thereIsNoChessNearby({x, y}))continue;
            if (m_map[x][y] != NO_CHESS)continue;
            int baseScore = evaluateOnePoint(isBlackNow, {x, y});//没有落子前的分数
            m_map[x][y] = isBlackNow ? BLACK_CHESS : WHITE_CHESS;
            int myScore = evaluateOnePoint(isBlackNow, {x, y});//我下这点我会得到的分数
            m_map[x][y] = isBlackNow ? WHITE_CHESS : BLACK_CHESS;
            int rivalScore = evaluateOnePoint(!isBlackNow, {x, y});//如果我不下这点则敌方会得到的分数
            m_map[x][y] = NO_CHESS;
            ret.push_back({{(myScore - baseScore) + (rivalScore - (-baseScore)), {x, y}});//要让我获益
最大 或者能让敌方获益最大的点下棋
        }
    }
    std::shuffle(ret.begin(), ret.end(), std::mt19937(std::random_device()()));
    std::sort(ret.begin(), ret.end(), [](const ScoreCoord &a, const ScoreCoord &b) {
        return a.score > b.score;
    });
    return ret;
}
```

性能测试



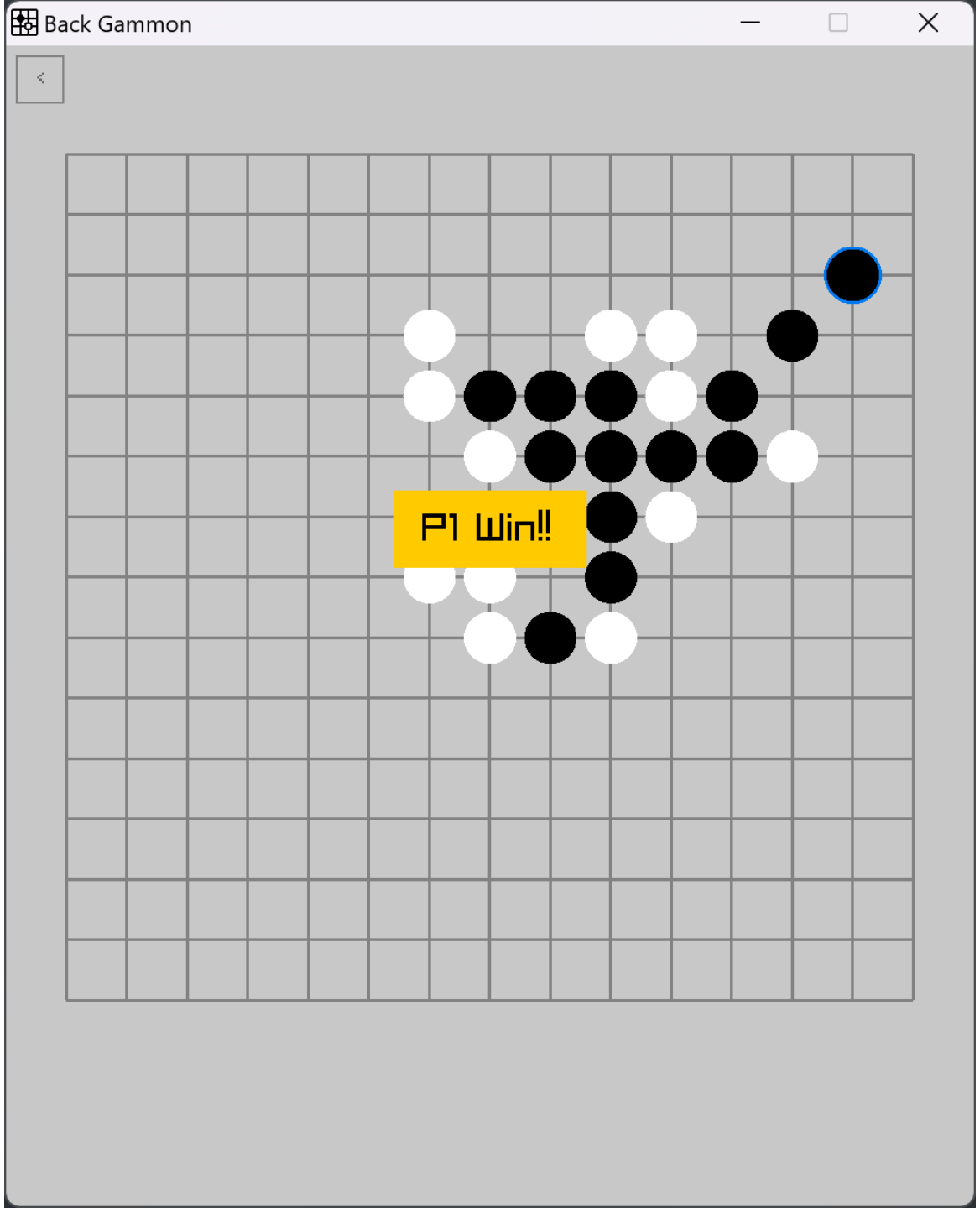
35603 items selected. | Slices (35603)

Name	Wall Duration	Self time	Average Wall Duration	Occurrences
static int ChessEngine::evaluateAll(bool)	184.778 ms	178.203 ms	1.004 ms	184
static int ChessEngine::evaluateOnePoint(bool, Coord)	103.885 ms	95.919 ms	0.026 ms	4041
static std::vector<ScoreCoord> ChessEngine::generatePossibleMove(bool)	97.339 ms	87.708 ms	4.056 ms	24
static int ChessEngine::getLineScore(const char*, bool)	17.587 ms	17.587 ms	0.001 ms	29412
static int ChessEngine::abSearch(int, int, int, bool, Coord&)	289.036 ms	16.012 ms	12.567 ms	23
static bool ChessEngine::someoneWin(Coord)	9.478 ms	9.375 ms	0.046 ms	207
static int ChessEngine::checkByStep(Coord, int, int)	0.114 ms	0.114 ms	0.000 ms	1712
Totals	**702.217 ms**	**404.918 ms**	**0.020 ms**	**35603**

Selection start	**6,404.997 ms**
Selection extent	**324.657 ms**

对战

- 在线对战（真人）先手和后手没有输过几次，大部分都是很快结束战斗了：（典型对局）
- AI能够准确的创造三三局面快速结束比赛



对战

- 不过和在线有难度的AI比起来还是偶尔会输，不过大部分是赢