

GGST格斗游戏源码核心模块分析

角色输入采集系统

- **底层输入读取**：游戏通过系统输入库（如 `sysue_InputPad.cpp`）读取手柄/键盘状态，将硬件按键映射到内部输入标志（`AA_CInput` 类）。对手柄输入，`AAUE_CInputPad` 负责获取摇杆、按键等原始数据；对键盘输入则有类似的类（`AAUE_CInputKeyBoard`）。这些输入设备对象被关联到角色的 `GAME_KeyController` 实例上。
- **按键环缓冲区**：每个角色拥有一个 `GAME_KeyController`（具体战斗模式下为 `GAME_BattleKeyController`）和对应的“键环缓冲区”（`m_KeyRingBuffer`）。每帧执行 `GAME_KeyController::Func()`，其内部调用 `UpdateKey()` 更新按键状态。`UpdateKey()` 会调用 `CheckOnTrigger()` 来获取当前按键“按下/弹起”状态，将结果填入当前帧的缓冲区节点，并在有按键变化时推进缓冲区索引。这样，输入历史以环形缓冲方式保存多帧的按键和定时信息。
- **按键状态标志**：角色使用 `U32` 类型的标志位记录当前按键状态，包括方向键（上、下、左、右）和动作键（P、K、S、HS、D等，内部对应 `RECFLG` 标志）。`m_TrgDown` / `m_TrgUp` 保存本帧新触发或释放的按键，`m_PureKey` 保存持续按下的按键。每推进一帧，`UpdateKey()` 会根据是否有新触发按键决定是否移动环缓冲区索引（`NextCurrent()`），并为新节点记录时间。缓冲区节点包含一个 `TrgTime`（持续时间）和按键数据。
- **控制器映射**：在 `GAMEUE_BattleKeyController::SetDefaultKeyTable()` 等函数中，游戏将常用方向或按键号（如 `GKN_UP`、`GKN_A`）映射为引擎底层按键（如 `UE_BUTTON_UP`、`UE_BUTTON_A`）。这样，系统将实际硬件输入自动转为游戏可识别的按键编号，再由 `CheckOnTrigger()` 收集进入缓冲区。

输入匹配逻辑

- **命令表定义**：游戏代码中定义了预设的技能命令序列，如在 `game_Key.cpp` 中通过 `COMMAND_SET` 数组描述各特殊技的输入（例如 `cmd_Hadou[]` 代表 236+P 波动拳）。每个元素包含 `{ GameKeyFlag, TrgCount }`，`GameKeyFlag` 中设置方向标志（如 `CMDKEY_F`、`CMDKEY_FD`、`CMDKEY_D` 等，见 `game_Key_Flag.h`）和可选标志位（`CMDKEY_TRG` 表示该帧需按下动作键，`CMDKEY_TAME` 表示充能/蓄力要求）。`TrgCount` 指定允许的时限。命令表通常以 `CMDKEY_END` 结尾。
- **输入缓冲分析**：`CBattleInputAnalyzer` 类维护了最近若干帧的方向输入序列（方向仅，去掉动作键），在每帧由角色逻辑更新：调用 `UpdateBattleInputAnalyzer(recFlg)` 将当前缓冲区顶点的方向标志加入记录，并递增每帧计时。
- **命令检测**：当角色更新其状态时，会调用 `CBattleInputAnalyzer::CheckCommand(CMD_KeyCommandID id, U32 frontDir)` 来检测命令。该函数从当前帧开始向过去追溯，依次比较 `COMMAND_SET` 表中定义的每一步需求与记录到的 `RECFLG` 输入。对于设置了 `CMDKEY_TRG` 的步骤，要求输入按下时对应按键；对于普通（无特殊标志）步骤，则只要求方向保持。算法会根据 `frontDir` 参数在面对左右时翻转方向（`FlipRecFlg` 实现）。若整个序列匹配且时间未超限，则返回真。
- **触发特殊指令**：在角色逻辑（如 `char_Base.cpp`）中，根据分析器结果触发技能。例如：`if (key->CheckCommand(CMD_Hadou, dir)) { m_InpFlag[dir][INP_HADOU] = true; }` 表示检测到 236+P 的输入并将波动拳标记为输入有效。类似地有 `CheckRotateCommand` 等检测 360 度旋转式命令，以及 `IsHighJump` 等快速判定高跳的便利接口。

连招与取消系统

- **取消条件标志**：角色对象包含多种取消相关标志和计时器，如 `m_CancelReq`（动作开始时允许取消）、`m_CancelDoneTimer`（取消成功后冷却）、`m_ForceRomanCancel` 等，这些由动作脚本初始化或

攻击命中时设置。`char_Actflag.h` 中还定义了各类状态标志，如角色是否处于攻击起始可取消帧（`PLATTACK_FLAG2`）等。

- **技能间取消**：当一个招式处于可取消窗口时，若玩家输入了下一个技能条件（例如连按符）并满足资源要求（如罗曼值充足），角色逻辑会提前结束当前招式并转入下一个动作。代码中，特定函数（例如 `char_Actsub.cpp` 中的 `CancelParamCtrl()`）会判断当前是否允许取消。完成特殊动作（如投掷、中段攻击）一般只能取消为其他特殊技，而普通轻/重击招在击中或防御帧后可取消为特殊技或闪回等。
- **连段窗口与判定**：系统通过记录攻击命中的时机来决定连段。举例：攻击击中对手（碰撞检测命中）后会设置一个“Cancel预约”标志（`m_CancelYoyakuTimer`）和更宽的取消窗口。此时只要后续输入符合技能要求（且前一技已击中），就可执行连招。脚本中也可能对特定动作设置仅在前摇或后摇某些帧取消。
- **资源与优先级**：罗曼值（Tension）等资源受 `m_TensionFlag`、`m_TensionLock` 等变量控制，只有资源足够时才能触发浪漫取消。不同取消类型（普通取消、浪漫取消、Just 浪漫）有不同优先级和消耗。系统确保同一时刻只执行一种取消（通常浪漫取消优先级高于普通取消）。

技能运行时逻辑

- **动作状态机**：技能启动后，角色进入对应的动作状态，由脚本或代码驱动帧推进。通常通过 `ActionBegin(skillName)` 开始一个技能，`ActionEnd()` 结束。动作脚本（BBS格式）定义了攻击帧数、碰撞盒、飞行轨迹等信息。每帧，角色执行 **帧推进**：更新 `m_Frame` 计数，并根据当前动作脚本读取相应帧的数据。
- **动画与骨骼**：角色的骨骼动画通过 `obj_BoneControl.cpp` 等模块来驱动。在每帧推进过程中，系统计算角色骨骼在该帧应有的旋转和平移变换，并更新角色网格的渲染姿态。动作脚本中定义的动画序列与角色骨骼绑定，从而实现图形化动作。
- **碰撞与命中检测**：在技能执行过程中，特定帧会激活攻击判定框（hitbox）。相关代码在碰撞模块（如 `obj_Collision.cpp` 或脚本解析模块 `obj_BBScript.cpp`）中执行。当角色脚本指令到达“产生碰撞判定”的帧时，会将攻击判定框数据（位置、大小、伤害等）插入碰撞检测系统。该系统每帧检查玩家是否与对手的攻击盒相交，并处理命中逻辑（扣血、击倒、硬直等）。
- **技能信息列表**：`char_Base` 包含 `CSkillInfoList m_SkillInfoList`，每执行一个技能就向该列表加入对应条目，包括技能分类、剩余帧等数据。引擎通过这些数据管理当前活跃技能和剩余时间，并在技能结束时清除。
- **游戏世界运动**：如果技能涉及位移（如前冲、跳跃），每帧还要更新角色位置。物理移动可能在脚本中给出速度向量（参见脚本内的移动参数），代码在角色更新时累加到角色位置上。

技能中断与结束

- **动作结束**：当技能脚本执行完毕（帧数用尽或到达 `ActionEnd` 指令）时，角色自动结束当前动作，返回空闲或待机状态。此过程常见于 `ActionEnd()` 调用后清理当前状态，重置各种计时器（如 `m_Frame=0`）、清空攻击判定。
- **被击中与中断**：如果角色在技能期间被对方攻击命中，通常会立即跳出当前动作，进入被击硬直状态。代码在碰撞处理中检测到命中后，会调用角色的“受击响应”逻辑（如 `Hit` 或 `Damage` 相关接口），并且根据攻击类型和剩余体力决定是否强制中断动作。
- **其他中断条件**：某些技能可被玩家主动中断，例如跳跃取消。游戏中当角色接收某些输入（如再按跳跃键）且符合同步条件时，会强制结束地面技并开始跳跃。类似地，防御取消（Faultless Defense）在防御瞬间可取消某些动作（通过检测 `GKF_L2` 键触发）。
- **被动停止**：动画完成、场景切换等也会结束技能。例如战斗结束时角色强制返回待机。代码中通常检查诸如胜负判定后清除所有行动请求。

配置表结构

- **技能表**：游戏中每个角色的技能数据可能由脚本或表格定义。虽源码中技能具体参数散见于脚本，工程中也可见 `REDDDataTable_SkillData` 这类表格，用于编辑器显示或辅助计算。角色类 `CSkillInfo` 存储

单个技能的动画长度、伤害、冲击力等。技能表字段包括技能编号、名称、分类（例如普通技、特殊技、超级技等）和相关参数。

- **输入命令表：**有多套输入配置：在 `game_Key.cpp` 中定义了基于数字小键盘（九宫格）方向的命令序列，如 `cmd_Hadou[]`。`game_Key_Cmdtbl.h` 还包含旧版“SNK式”命令表。`game_Key_Flag.h` 定义了所有可能的单步输入标志（`CMDKEY_*`）。角色执行命令匹配时对应这些表进行比对。
- **角色状态表：**状态方面，`char_Actflag.h` 中枚举了角色可能的运动标志（如蹲下、倒地、受击等）。此外，`PLAYER_FLAG` 系列标志定义角色的站立/蹲下/跳跃等大状态。动作脚本里通过命令（`ActionBegin/End`）控制状态过渡，条件判断（如冲刺、跳跃）则参考这些标志位。
- **配置绑定：**以上表格和枚举在代码中有对应的绑定。例如，命令表中 `CMD_Hadou` 对应的枚举值在角色逻辑里与技能名称关联（通常在角色初始化或脚本中将命令ID映射到技能ID）。状态标志在角色更新时自动维护；技能表的数据最终填充到 `CSkillInfo` 或脚本节点里影响实际游戏逻辑。

以上分析基于提供的源码模块展开，涵盖了输入采集、命令匹配、连招取消、技能执行与中断、以及配表结构等方面的关键机制和流程。
