Tekken DSL - Technical Implementation Document

Overview

This document describes the C++ techniques used to implement the Tekken DSL. The DSL compiles as valid C++ code through clever use of preprocessor macros, operator overloading, and temporary objects.

Core Techniques

1. Preprocessor Macros for Keywords

All DSL keywords are C preprocessor macros that expand into valid C++ code.

```
#define BEGIN_GAME int main() { resetGame(); if(false){}else if((_FInit_{"_DUMMY_","Rushdown
#define END_GAME ,false)){} return 0; }
#define CREATE ,false)){}else if((
#define FIGHTER _FInit_
#define ABILITY _AInit_
```

The If-Chain Pattern: The semicolon-free syntax is achieved using an if-else chain pattern:

```
if(false){}
else if((expression1, false)){}
else if((expression2, false)){}
...
```

Each `CREATE` statement becomes an `else if` clause. The comma operator evaluates the expression (registering the fighter/ability) and returns `false`, so the chain continues.

---

2. Operator Overloading for Space-Separated Commands

Commands like `DAMAGE DEFENDER 22` use `operator<<` chaining:

```
#define DAMAGE ; _DmgCmd_{_attacker_, _round_} <<
#define DEFENDER _defender_ <<

// DAMAGE DEFENDER 22 expands to:
// ; _DmgCmd_{_attacker_, _round_} << _defender_ << 22
```

Helper Structs:

```
struct _DmgCmd_ {
    Fighter& attacker;
```

```
    int round;
    _DmgFinal_ operator<<(Fighter& target) const {
        return _DmgFinal_(target, attacker, round);
    }
};

struct _DmgFinal_ {
    Fighter& target;
    Fighter& attacker;
    int round;
    void operator<<(int dmg) const {
        int final_dmg = static_cast<int>(dmg * attacker.getOutgoingMod(target, round)
                                            * target.getIncomingMod(attacker));
        target.takeDamage(final_dmg);
    }
};
```

The chain: `_DmgCmd_` → `operator<<(Fighter&)` → `_DmgFinal_` → `operator<<(int)` → execute damage

---

3. The Comma Operator for Collections

The `operator,` is overloaded to collect multiple items:

```
struct _FighterCollector_ {
    mutable std::vector<_FInit_> entries;

    _FighterCollector_& operator,(_FInit_ f) {
        entries.push_back(f);
        return *this;
    }

    operator bool() const {
        finalize();  // Register all fighters
        return false;
    }
};
```

This allows:

```
CREATE FIGHTERS [
    FIGHTER {...},
    FIGHTER {...}
]
```

---

4. Lambda Capture for Deferred Execution

Ability actions are stored as `std::function` lambdas:

```
#define ACTION 0 ? (AbilityAction)nullptr
#define START [&](Fighter& _attacker_, Fighter& _defender_, int _round_, ActionContext& _ctx
    (void)_round_; (void)_ctx_; int _d_=0; {
#define END ;}}
```

The `ACTION: START ... END` block becomes a lambda that captures the ability's code and executes it during battle.

---

5. Scheduled Actions (FOR/AFTER)

The `ActionContext` class manages scheduled actions:

```
class ActionContext {
    std::vector<std::pair<int, std::function<void()>>> forActs_, afterActs_;
    int round_ = 0;
public:
    void scheduleFor(int r, std::function<void()> a);    // Execute for N rounds
    void scheduleAfter(int r, std::function<void()> a); // Execute after N rounds
    void processRound(int r);                            // Called each round
};
```

FOR Scheduler:

```
struct _ForScheduler_ {
    ActionContext* ctx;
    int rounds;
    Fighter* attacker;
    Fighter* defender;

    template<typename F>
    void operator=(F&& action) {
        ctx->scheduleFor(rounds, [=]() mutable {
            ActionContext dummyCtx;
            action(*attacker, *defender, 0, dummyCtx);
        });
    }
};

#define FOR ; _ForScheduler_{&_ctx_,
#define ROUNDS , &_attacker_, &_defender_} = [&](Fighter& _attacker_, Fighter& _defender_, i
```

---

6. The TAG Command with Greek Alpha

The TAG command uses the Greek letter (alpha) for exit:

```
struct _TagAlphaValue_ {
    int dummy;
    _TagAlphaValue_ operator-() const { return *this; }  // Handle ---
    _TagAlphaValue_& operator--() { return *this; }
};

static _TagAlphaValue_  ;  // Greek alpha variable

struct _TagFinal_ {
    Fighter& target;
    void operator<<(const _TagAlphaValue_&) const {
        target.setInRing(false);  // Exit ring
    }
    void operator<<(const _TagUnderscoreValue_&) const {
        target.setInRing(true);   // Enter ring
    }
};
```

`TAG DEFENDER ---` expands to operators that eventually call `setInRing(false)`.

---

7. DEAR...LEARN Syntax

The ability learning syntax uses `operator[]`:

```
struct _LearnOp_ {
    const char* fighterName;

    bool operator[](_AbilityLearnCollector_ c) const {
        Fighter& f = getFighter(fighterName);
        for (const auto& name : c.names) {
            f.addAbility(name);
        }
        return false;
    }
};

#define DEAR ,false)){}else if((_LearnOp_(
#define LEARN )
#define ABILITY_NAME(x) + _AbilityNameAdder_(#x)
```

The `+` operator chains ability names without commas:

```
ABILITY_NAME(Ability1)
```

```
ABILITY_NAME(Ability2)
// Becomes: + _AbilityNameAdder_("Ability1") + _AbilityNameAdder_("Ability2")
```

---

8. Getter Functions with Proxy Objects

```
struct _GetHPProxy_ {
    Fighter* f;
    _GetHPProxy_& operator<<(Fighter& fighter) { f = &fighter; return *this; }
    int operator<<(_GetEnd_) const { return f ? f->getHP() : 0; }
};

#define GET_HP(x) (_GetHPProxy_{} << x _get_end_)
```

`GET_HP(DEFENDER)` expands to a proxy chain that returns the HP value.

---

9. Logical Operators (AND/OR/NOT)

Variadic macros with immediately-invoked lambdas:

```
#define AND(...) ([&]{ bool _args_[] = {__VA_ARGS__}; \
    for(bool _b_ : _args_) if(!_b_) return false; return true; }())

#define OR(...) ([&]{ bool _args_[] = {__VA_ARGS__}; \
    for(bool _b_ : _args_) if(_b_) return true; return false; }())

#define NOT(x) (!(x))
```

The `[&]` capture ensures references work correctly in nested conditions.

---

10. Control Flow (IF/ELSE/END)

```
#define IF ;{if(
#define DO ){
#define ELSE ;}else{
#define ELSE_IF ;}else if(
#define END ;}}
```

These expand to standard C++ if-else blocks wrapped in braces for scoping.

Fighter Type Damage Modifiers

Implemented in the `Fighter` class:

```
double getOutgoingMod(const Fighter& target, int round) const {
    switch(type_) {
        case FighterType::Rushdown:
            return (target.type_ == FighterType::Grappler) ? 1.20 : 1.15;
        case FighterType::Evasive:
            return 1.07;
        case FighterType::Grappler:
            return (round % 2 == 1) ? 1.07 : 1.0;
        default: return 1.0;
    }
}

double getIncomingMod(const Fighter& attacker) const {
    switch(type_) {
        case FighterType::Heavy:
            return (attacker.type_ == FighterType::Evasive) ? 0.70 : 0.80;
        case FighterType::Evasive:
            return 0.93;
        default: return 1.0;
    }
}
```

Key Design Decisions

1. **Header-Only**: Everything in `Tekken.h` for easy inclusion
2. **Global Registries**: `g_fighters()` and `g_abilities()` store game state
3. **Semicolon-Free**: If-chain pattern eliminates need for semicolons
4. **Type Safety**: Proxy objects ensure correct operator chaining
5. **UTF-8 Support**: Greek character requires UTF-8 source encoding

---

Build Requirements

- C++11 or later (for lambdas, initializer lists)
- UTF-8 source file support
- Standard library: `<string>`, `<vector>`, `<map>`, `<functional>`, `<iostream>`