Метапрограммирование для игрового движка

Антон Дунчев

Engine developer,
Wargaming

Игра

Контент:

- Текстуры
- Модели
- Анимации
- Системы частиц
- Описание игровых объектов
- Шейдеры
- Скрипты
- ...

Движок:

- Виртуальная машина интерпретации контента и пользовательского ввода

Часть 1. Python в World of Tanks

- CPython\API 2.7.7
- Legacy интеграция. (5+ лет)
- Виртуальная машина в main-потоке
- Script::tick() - 25% времени кадра
- >100 C++ wrapper классов интегрированных в скрипты
- >1000 методов\функций экспортированных в Python

Экспорт C++ функций в Python


```
static PhysBody* constructPhysicsBody(
    WorldId id,
    const Matrix& transform,
    const Vector3& initialImpulse,
    const PhysicsInfo& info);
```

Экспорт C++ функций в Python


```
PY_AUTO_EMBED_FUNCTION(RETDATA, example, constructPhysicsBody,
    ARG(WorldId,
        ARG(Matrix,
            ARG(Vector3,
                ARG(PhysicsInfo, END)))))
```

Альтернативы

- PyBind11
- Boost::Python

PyBind11

Плюсы:

- Modern C++
- Python 2/3
- Header-only
- Шаблоны
- Без лишних зависимостей

Минусы:

- Небольшое комьюнити
- Шаблоны
- Сложности интеграции в старый код

Boost::Python

Плюсы:

- Большое комьюнити
- Несложная интеграция

Минусы:

- Зависимость от половины библиотеки Boost
- Не header-only
- Macros
- Время компиляции

Экспорт функций в Python

- Маршалинг типов
- Обработка исключений
- Экспорт опциональных аргументов

Значение опционального аргумента - не часть типа


```
int foo(int i, int j = 1);
int bar(int i, int j);
static_assert(std::is_same_v<decltype(foo), decltype(bar)>, "Equal");
```

Экспорт опциональных аргументов

BigWorld: Рекурсивные макросы

```
int add(int i, int j = 5) {
    return i + j;
}
PY_AUTO_EMBED_FUNCTION(RETDATA, example, add,
    ARG(int,
        OPTARG(int, 5, END)))
```

Экспорт опциональных аргументов

Boost::Python: Перегрузка функции

```
int add(int i, int j = 5) {
    return i + j;
}

using namespace boost::python;
BOOST_PYTHON_FUNCTION_OVERLOADS(add_overloads, add, 1, 2)
BOOST_PYTHON_MODULE(example) {
    def("add", add, add_overloads());
}
```

Экспорт опциональных аргументов

PyBind11: дублирование информации

```
int add(int i, int j = 5) {
    return i + j;
}

namespace py = pybind11;
PYBIND11_EMBEDDED_MODULE(example, m) {
    m.def("add", &add, "A function which adds two numbers",
        py::arg("i"), py::arg("j") = 5);
}
```

Что общего между подходами?

```
int add(int i, int j = 5) {
    return i + j;
}
```

Неизменная декларация экспортируемой функции

Реальное положение вещей

На каждый C++ класс пишется свой CPython класс - обертка.

Экспортированные функции изолированы от нативной реализации.

Экспортированные функции могут содержать дополнительную логику обработки переданных аргументов.

Тогда почему бы не интегрировать информацию о значении опционального аргумента в тип?

Опциональный аргумент

Non-type template parameter

```
template<typename T, T default_value>
struct opt_carg
{
    T value = default_value;

    constexpr opt_carg() = default;
    constexpr opt_carg(T val) : value(std::move(val)) {}
    constexpr operator T() { return value; }
    constexpr operator const T() const { return value; }
    constexpr T& operator*() { return value; }
    constexpr const T& operator*() const { return value; }
};
```

Реализация

Ленивое вычисление опционального аргумента

```
template<typename T, T default_value()>
struct opt_arg
{
    T value = default_value();

    constexpr opt_arg() = default;
    constexpr opt_arg(T val) : value(std::move(val)) {}
    constexpr operator T() { return value; }
    constexpr operator const T() const { return value; }
    constexpr T& operator*() { return value; }
    constexpr const T& operator*() const { return value; }
};
```

Экспорт опциональных аргументов

Encore: значение опционального аргумента - часть типа

```
int add(int i, opt_carg<int, 5> j = {});
PY_EMBED_MODULE_FUNCTION(RETDATA, "example", add);

static int add_j_default() { return 5; }
int sub(int i, opt_arg<int, add_j_default> j = {});
PY_EMBED_MODULE_FUNCTION(RETDATA, "example", sub);
```

Использование


```cpp
int main(int argc, char* argv[]) {
    using namespace std;
    cout << "C++: " << add(42) << "\n";
    cout << "C++: " << add(40, 2) << "\n";

    Py_Initialize();
    Py_InitModule("example", s_exampleMethods);
    PyRun_SimpleString(R"(
import example
print 'Python: ', example.add(42)
print 'Python: ', example.add(40, 2)
)");
    Py_Finalize();
    return 0;
}
```

```
C++: 47
C++: 42
Python:  47
Python:  42
```

Что получилось?

- Информация о значении опционального аргумента - часть типа.
- Ad-hoc решение, без необходимости переписывать уже написанный код.
- Простое описание экспортируемых функций.
- Возможность рефакторинга только в 1 месте в C++.
- Возможность менять значение по умолчанию без изменения декларации или тела функции

Часть 2. Материалы

Материал: шейдер + параметры

Специальный язык описания шейдеров WgFX:

- транслируется в HLSL (язык шейдеров DirectX)
- агрегирует все трансформации шейдера
  (1 шейдер WgFX = N шейдеров HLSL выбираемых на рантайме)
- предоставляет описание всех uniform буферов

Кеширование хендлов. Использование


```
void applyParameters(ManagedEffect* effect) {
    effect->setParameter("tintMap", m_tintMapTexture);
    effect->setParameter("exclusionMap", m_exclusionMapTexture);
    effect->setParameter("tiling", m_tiling);
}
```

String-typed programming

Связь данных через строки.
Разработчики привыкли оперировать названиями.
Операции со строками - долго.
Оффсеты - быстро, но не удобно.

Хендлы параметров

Описание параметра:

- Название параметра
- Тип параметра (размер данных)
- Смещение в uniform-буфере
- Набор атрибутов

Использование хендлов


```
void CamoHandler::prepareEffect(ManagedEffect* effect) {
    m_tintMapHandle = effect->getParameterByName("tintMap");
    m_exclusionMapHandle = effect->getParameterByName("exclusionMap");
    m_tilingHandle = effect->getParameterByName("tintTiling");
}
```

```
void CamoHandler::applyParameters(ManagedEffect* effect) {
    effect->setParameter(m_tintMapHandle, m_tintMapTexture);
    effect->setParameter(m_exclusionMapHandle, m_exclusionMapTexture);
    effect->setParameter(m_tilingHandle, m_tiling);
}
```

Кеширование хендлов. Реализация


```
template<const char*... Names>
class CachedParamHandles
{
    static constexpr size_t COUNT = sizeof...(Names);
    struct EffectData
    {
        std::array<ParamHandle, COUNT> handles = {};
        uint32_t effectGen = 0;
    };
    std::unordered_map<const ManagedEffect*, EffectData> m_handleMap;
```

Кеширование хендлов. Реализация


```
template<size_t Index>
ParamHandle handle(const ManagedEffect* effect)
{
    static_assert(Index < COUNT, "Index is out of bounds for CachedParamHandles");
    if (auto it = m_handleMap.find(effect);
        it != m_handleMap.end() && it->second.effectGen == effect->compileMark())
    {
        return it->second.handles[Index];
    }
    m_handleMap[effect].handles = {effect->getParameterByName(Names)...};
    m_handleMap[effect].effectGen = effect->compileMark();
    return m_handleMap[effect].handles[Index];
}
```

Кеширование хендлов. Реализация


```
template<const char*... Names>
class CachedParamHandles
{
    static constexpr size_t COUNT = sizeof...(Names);
public:
    template<size_t Index>
    ParamHandle handle(const ManagedEffect* effect)
    {
        static_assert(Index < COUNT, "Index is out of bounds for CachedParamHandles");
        if (auto it = m_handleMap.find(effect);
            it != m_handleMap.end() && it->second.effectGen == effect->compileMark())
        {
            return it->second.handles[Index];
        }
        m_handleMap[effect].handles = {effect->getParameterByName(Names)...};
        m_handleMap[effect].effectGen = effect->compileMark();
        return m_handleMap[effect].handles[Index];
    }

private:

    struct EffectData
    {
        std::array<ParamHandle, COUNT> handles = {};
        uint32_t effectGen = 0;
    };
    std::unordered_map<const ManagedEffect*, EffectData> m_handleMap;
};
```

Прикладной код


```
private:
    struct ExposedFields {
        enum {
            TINT_MAP = 0,
            EXCLUSION_MAP,
            TILING,
            ROTATION
        };
    };
    static constexpr char CAMO_TINT_MAP[] = "tintMap";
    static constexpr char CAMO_EXCLUSION_MAP[] = "exclusionMap";
    static constexpr char CAMO_TILING[] = "tintTiling";

    CachedParamHandles<
        CAMO_TINT_MAP,
        CAMO_EXCLUSION_MAP,
        CAMO_TILING> m_handles;
```

Кеширование хендлов. Использование


```
void applyParameters(ManagedEffect* effect) {
    effect->setParameter(
        m_handles.handle<ExposedFields::TINT_MAP>(effect),
        m_tintMapTexture);

    effect->setParameter(
        m_handles.handle<ExposedFields::EXCLUSION_MAP>(effect),
        m_exclusionMapTexture);

    effect->setParameter(
        m_handles.handle<ExposedFields::TILING>(effect),
        m_tiling);
}
```

Кеширование хендлов. Выводы

Плюсы:

- Производительность приложения

Минусы:

- Необходимость поддерживать порядок значений перечисления и порядок имен
- Нельзя писать имя внутри параметров инстанцирования шаблона

Что поможет?

Кодогенерация.

Минусы:

- Нужна кастомная билд-система
- Нет стандартного подход, каждый решает задачу по разному

Плюсы:

- Решает проблему
- Реализуема прямо сейчас

Что поможет?

Метаклассы Херба Саттера.

Минусы:
- C++23 (возможно)

Плюсы:
- Решает проблему
- Стандартный поход

Часть 3. Передача аргументов функции

- Большое количество систем
- Множество настроек
- Установка настроек - атомарная операция

```
bool initSwapchain(
    int width,
    int height,
    int numRenderTargets,
    int fillColor,
    bool recreate = false);
```

Передача аргументов в структуре


```
void foo(int a, float b, double c);

struct params
{
    int a;
    float b;
    double c;
};
void bar(params p) {
    auto [a, b, c] = p;
    // ...
}
```

Именованные аргументы?


```
void foo(int a, float b, double c);

struct params
{
    int a;
    float b;
    double c;
};
void bar(params p) {
    auto [a, b, c] = p;
    // ...
}
```

```
int main() {
    bar({.a = 1});

    return 0;
}
```

Опциональные аргументы


```
struct params {
    int a;
    float b = 0.0f;
    double c = c_default();

private:
    static double c_default() { return 1.0; }
};
void bar(params p) {
    auto [a, b, c] = p;
    // ...
}
```

Проблема

Обязательные аргументы

```
int main() {
    bar({});

    return 0;
}
```

Решение - opt_arg наоборот

Тип без конструктора по умолчанию

```
template<typename T>
struct required
{
    T value;

    constexpr required() = delete;
    constexpr required(T val) : value(std::move(val)) {}
    constexpr operator T() { return value; }
    constexpr operator const T() const { return value; }
    constexpr T& operator*() { return value; }
    constexpr const T& operator*() const { return value; }
};
```

Все вместе


```
struct params {
    required<int> a;
    float b = 0.0f;
    double c = c_default();

private:
    static double c_default() { return 1.0; }
};
void bar(params p) {
    auto [a, b, c] = p;
    // ...
}

int main() {
    bar({1, .b = 2.0});

    return 0;
}
```

Выводы

Для игровой индустрии в C++ важны перформанс и верхнеуровневые zero-overhead абстракции.

Метапрограммирование - это мощный инструмент для рефлексии как игрового контента, так и структур самого языка.

Там где не справляются шаблоны обходимся кодогенерацией и C++ препроцессором.

Даешь рефлексию в C++!

Хардкорный бонус

Как это работает?

Image credits: Discovery Communications

function_traits


```cpp
template<class R, class... Args>
struct function_traits<R(Args...)>
{
    using return_type = R;

    static constexpr size_t arity = sizeof...(Args);

    template <size_t N>
    struct argument
    {
        static_assert(N < arity, "error: invalid parameter index.");
        using type = typename std::tuple_element_t<N,std::tuple<Args...>>;
    };

    template <size_t N>
    using arg_t = typename argument<N>::type;
};
```

Немножко шаблонной магии


```
template<typename R, typename... Args, size_t... Idx>
auto convert_all(std::index_sequence<Idx...>, const PyObject* pyTuple)
    -> std::optional<std::tuple<Args...>>
{
    using traits = function_traits<R(Args...)>;
    auto retVal = std::make_tuple(
        convert_helper<typename traits::template arg_t<Idx>>::convert(PyTuple_GetItem(pyTuple, Idx))...);
    auto ok = (std::get<Idx>(retVal).has_value() && ...);
    if(!ok) {
        return std::nullopt;
    }
    return std::make_tuple(std::move(std::get<Idx>(retVal).value())...);
}

template<class R, class... Args>
R call_from_python(R(*fn)(Args...), const PyObject* pyTuple) {
    return std::apply(fn,
        *convert_all<R, Args...>(std::index_sequence_for<Args...>(), pyTuple));
}
```

Щепотка хелперов


```cpp
template<typename T>
struct convert_helper
{
    static std::optional<T> convert(const PyObject* arg)
    {
        auto val = T{};
        if(setData(val, arg))
        {
            return val;
        }
        return std::nullopt;
    }
};

template<typename T, T default_value()>
struct convert_helper<opt_arg<T, default_value>>
{
    static std::optional<opt_arg<T, default_value>> convert(const PyObject* arg)
    {
        if(contains_value(arg))
        {
            return convert_helper<T>::convert(arg);
        }
        return opt_arg<T, default_value>{};
    }
};
```