Making the World Playful

Modern C++ In The Fiction Factory Engine

Real case example of Modern C++ usage in game engine and tools development

Lucio Asnaghi - Fiction Factory 2023

What is Fiction Factory?

Fiction Factory Is:

- Internal game engine powering **King** games.
- Provides both **Runtime** (mobile/web) and **Editor** (desktop) applications.
- Set of tools for building / deploying games code and assets.
- It heavily relies on **C++**. Currently on C++17 but moving to C++20 soon.

A Special Mention: KSTL

At the heart of our C++ stack lives **King Standard Library**:

- It is a comprehensive C++ library of classes, functions and tools.
- It was created to bridge the gap with (and improving) upcoming standards.
- It is completing and not replacing the C++ standard library.

C++ In Action: The Problem

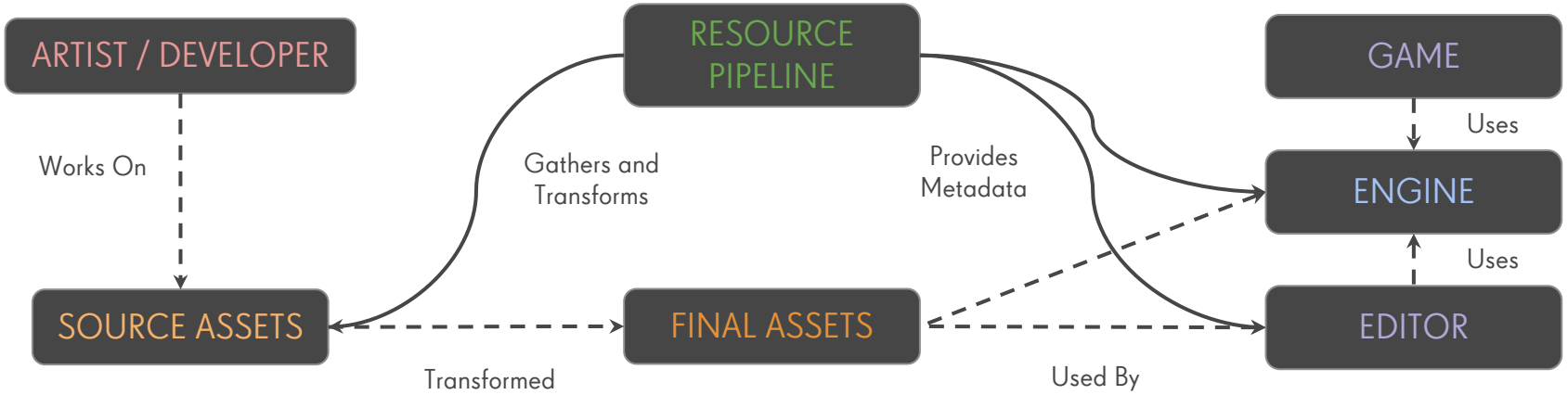
Candycrush Saga has more than 10 years:

- It contains a reasonably sized set of packages / assets.
- To work with those assets, they need to be **gathered** and **scanned**.
  - Scanning assets is slow.
- Some of those assets needs to be **processed** (encoded) to be used.
  - Encoding assets takes time.
- A frequent operation like opening a project can take considerable amount time.

The Resource Pipeline

A **resource pipeline** is a tool that:

- Provides the filesystem tree of the assets available in the project.
- Categorizes assets based on their type and metadata.
- Transform assets into their final "shape": for example **.wav** -> **.mp3**



The Initial Picture: Assets Gathering

Elapsed Time (seconds)

Candycrush Saga Project Gathering: *320 packages / ~100k assets*

The Mission Objectives

- Categorizing assets **efficiently**.
- Providing tooling to **simplify and accelerate** working with assets.
- **Replace** the resource pipeline when fast iteration is needed.

The Key For Speed: Principles

- Use C++ all down to the last bit.

- Don't execute if you don't need that information:
  - Open once, and memory map file accesses.
  - Avoid whole file scan and whole xml/json parsing.
  - Parse ondemand when possible, early return.

- Improve data layout:
  - Use hash maps/sets for fast lookup by filter query, O(1) lookups.
  - Linear searches are bad, have worse case O(N).
  - Use binary searches, with worse case O(logN).

- Everything should be immutable (from the API perspective).

- Parallelize all the inspection work.

Memory map when it speeds up !

Use memory mapping with sequential advise, to speed up filesystem accesses: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2044.html

```
class memory_mappable
{
public:
   typedef /*implementation-defined*/ accessmode_t;
   typedef /*implementation-defined*/ mapping_handle_t;
   static const mapping_handle_t      invalid_handle;
   static const accessmode_t          read_only;
   static const accessmode_t          read_write;
   static const accessmode_t          invalid_mode;

protected:
   void init(mapping_handle_t handle, accessmode_t mode);

public:
   memory_mappable();

   memory_mappable(memory_mappable &&other);

   mapping_handle_t get_handle() const;

   accessmode_t get_mode() const;

   virtual ~memory_mappable() = 0;
};
```

```
class mapped_region
{
public:
   typedef /*implementation-defined*/ offset_t;
   typedef /*implementation-defined*/ accessmode_t;
   static const accessmode_t          invalid_mode;
   static const accessmode_t          read_only;
   static const accessmode_t          read_write;
   static const accessmode_t          copy_on_write;

   mapped_region();

   mapped_region(const memory_mappable& mappable,
                 accessmode_t           mode,
                 offset_t               mappable_offset,
                 std::size_t            size = 0,
                 const void *           address = 0);

   mapped_region(mapped_region &&other);

   std::size_t  get_size() const;

   void*        get_address() const;

   offset_t     get_offset() const;

   accessmode_t get_mode() const;

   void flush(std::size_t region_offset = 0, std::size_t numbytes = 0);

   void swap(mapped_region &other);

   ~mapped_region();
};
```

Memory map... then use spans to avoid copies !

```cpp
// Create a file_mapping object
king::file_mapping  mapping("/home/user/file", king::memory_mappable::read);

// Create a mapped_region covering the whole file
king::mapped_region region(mapping, king::mapped_region::read);

// Obtain the size and the address of the mapped region
void* address    = region.get_address();
std::size_t size = region.get_size();
```

```cpp
// Convert the mapped region to a span
auto file_data = king::span<std::byte>(address, size);

// Construct an input stream over the span
king::ispanstream stream { file_data };
```

std::basic_ispanstream

Defined in header <spanstream>

```cpp
template<
    class CharT,
    class Traits = std::char_traits<CharT>    (since C++23)
> class basic_ispanstream
    : public basic_istream<CharT, Traits>
```

The class template `std::basic_ispantream` implements input operations on streams based on fixed buffers.

Don’t fully parse, parse on demand instead !

As we need to categorize .xml files, do it efficiently.

```
king::optional<asset_type> xml_to_asset_type(const scan_context& context, std::istream& file_stream, const fs::path& xml_path)
{
    king::expect(file_stream.good());

    file_stream.seekg(0, std::ios_base::beg);

    king::optional<std::string> tag_name;

    try
    {
        xml::parser parser(file_stream, xml_path.string());

        for (xml::parser::event_type e : parser)
        {
            if (e == xml::parser::start_element)
            {
                tag_name = parser.name();
                break;
            }
        }
    }
    catch (const xml::parsing& e)
```

Don’t linear search, insert sorted and binary search !

Sorted Insertions (Slower than push back).

```
auto it = king::ranges::lower_bound(target, asset_to_insert, &predicate_less<asset>);
if (it == target.end() or *it != asset_to_insert)
    target.insert(it, asset_to_insert);
```

Quick Lookups.

```
auto first = king::ranges::lower_bound(assets_span, parent_path.native(), king::ranges::less{}, proj);
if (first == assets_span.end())
    return {};
```

Quick removals.

```
auto it = king::ranges::lower_bound(target, asset_to_remove, &predicate_less<asset>);
if (it != target.end() and *it == asset_to_remove)
    target.erase(it);
```

Don’t copy containers, iterate ranges !

Use modern language features to simplify code and retain performance.

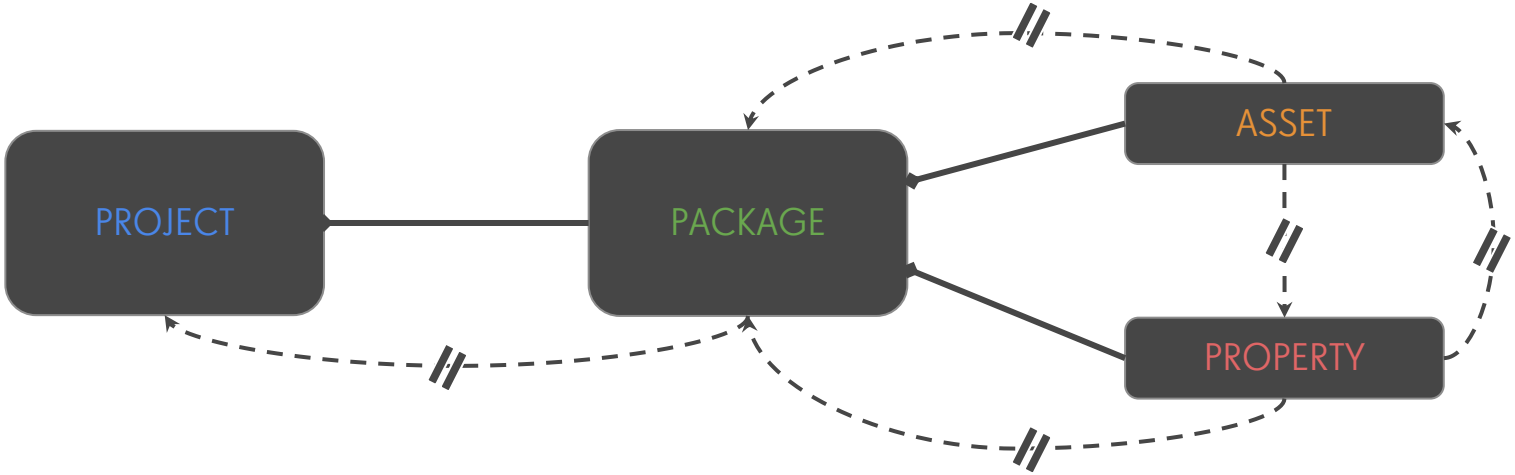
```
inline auto project::get_assets() const
{
    return packages_
        | king::ranges::views::transform([](const auto& pkg) { return pkg->get_assets(); })
        | king::ranges::views::join;
}
```

```
inline auto project::get_assets_by_type(asset_type type) const
{
    return packages_
        | king::ranges::views::transform([type](const auto& pkg) { return pkg->get_assets_by_type(type); })
        | king::ranges::views::join;
}
```

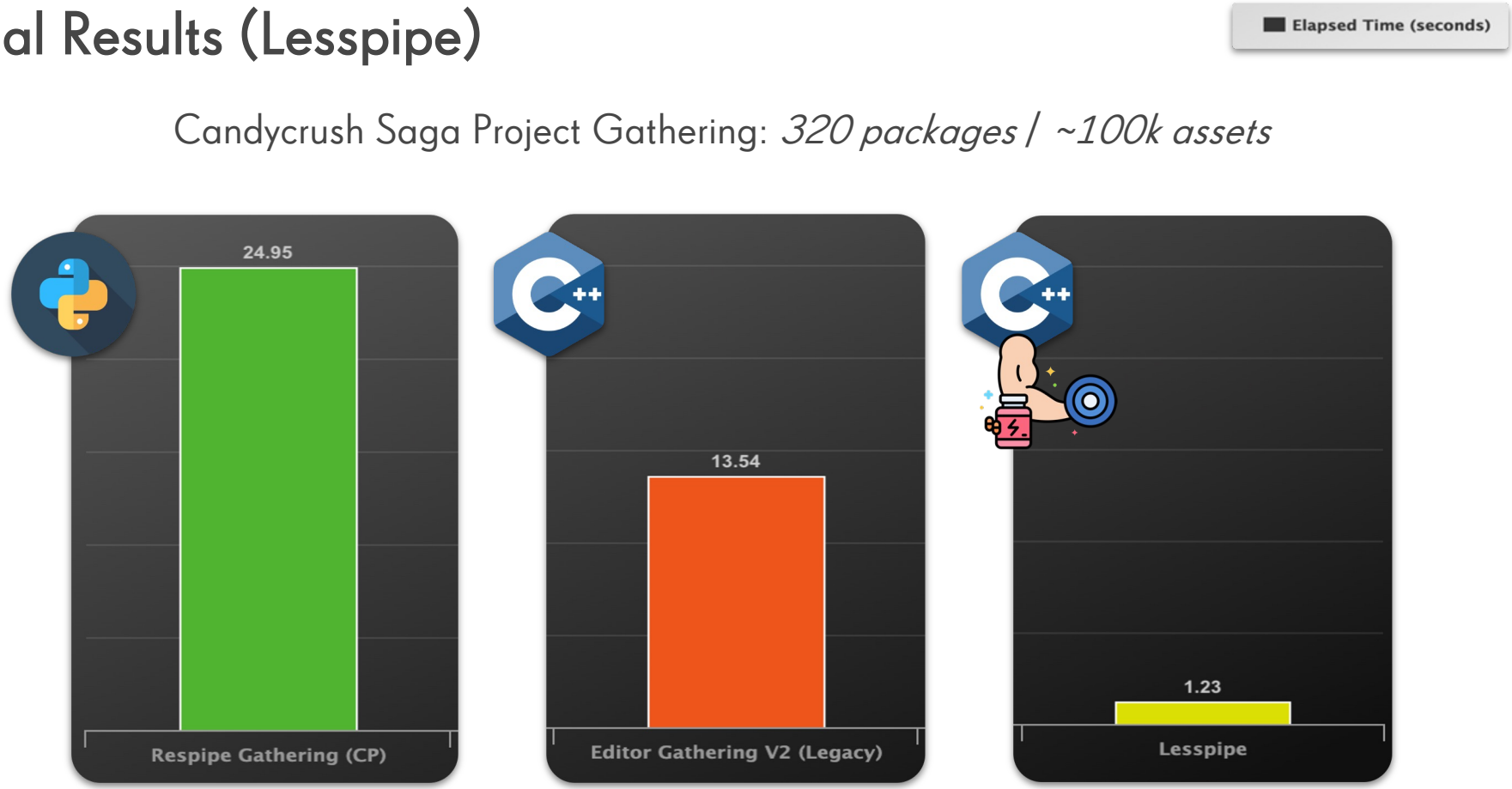
Children are agnostic of parents / siblings

Benefits of the approach:

- You can only modify children.
- The state of the parent is immutable from the POV of the child.
- Parallelization is then possible way down to the leafs.



Final Results (Lesspipe)



Lucio Asnaghi - Fiction Factory

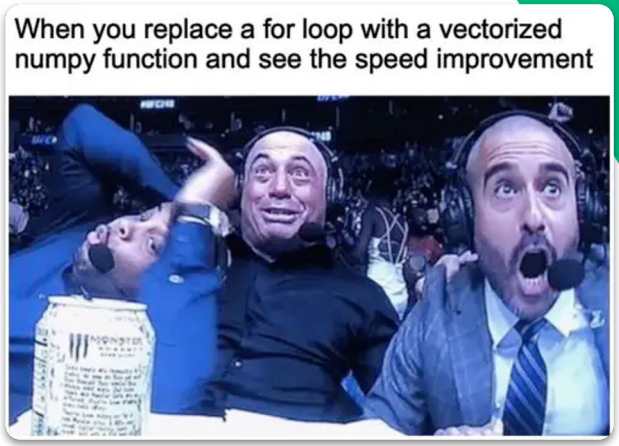
Learnings

More Learnings

- Use the right tools for the job.

- Avoid copies, embrace iterators/ranges, always reason in term of algorithmic complexity.

- You need to be fast ? Optimize your design. Avoiding early Premature Optimization is not preventing having an Optimized Design early.

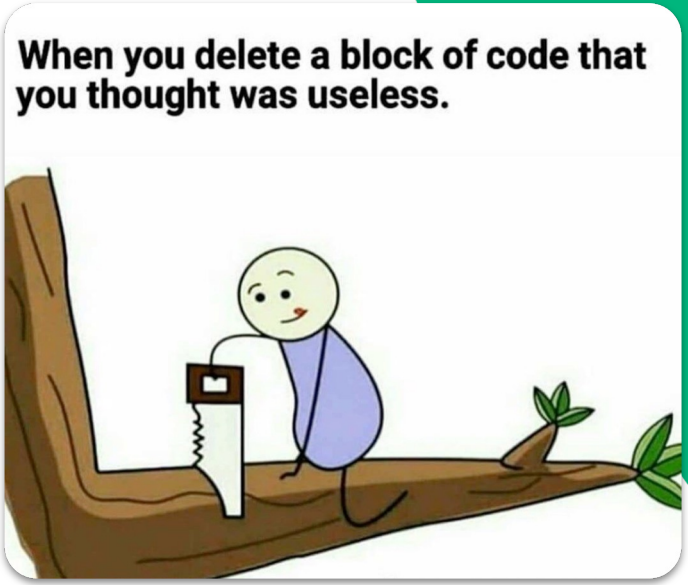


Even More Learnings

- Having it working doesn't mean it's finished. You overlooked the last 10% which takes 90% of the run time.

- Isolate complexity into tools/libraries. It's easier to optimize and focus just on the single task, with clearer separation of concerns.

- Designing APIs correctly is key to convert Code into Tech Capital. Take your time and do it right!



Thank You !