Program Listing for File codec_registry.hpp

Return to documentation for file (include/holoscan/core/codec_registry.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2023-2024 NVIDIA CORPORATION & AFFILIATES. All rights reserved. * SPDX-License-Identifier: Apache-2.0 * * Licensed under the Apache License, Version 2.0 (the "License"); * you may not use this file except in compliance with the License. * You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License. */ #ifndef HOLOSCAN_CORE_CODEC_REGISTRY_HPP #define HOLOSCAN_CORE_CODEC_REGISTRY_HPP #include <complex> #include <functional> #include <memory> #include <string> #include <type_traits> #include <typeindex> #include <typeinfo> #include <unordered_map> #include <utility> #include <vector> #include "./codecs.hpp" #include "./common.hpp" #include "./errors.hpp" #include "./expected.hpp" #include "./message.hpp" #include "./type_traits.hpp" #include "gxf/core/expected.hpp" #include "gxf/serialization/serialization_buffer.hpp" using std::string_literals::operator""s; namespace holoscan { class CodecRegistry { public: using GXFEndpoint = nvidia::gxf::Endpoint; using SerializeFunc = std::function<nvidia::gxf::Expected<size_t>(const Message&, GXFEndpoint*)>; using DeserializeFunc = std::function<nvidia::gxf::Expected<Message>(GXFEndpoint*)>; using Codec = std::pair<SerializeFunc, DeserializeFunc>; inline static SerializeFunc none_serialize = []([[maybe_unused]] const Message& message, [[maybe_unused]] GXFEndpoint* buffer) -> nvidia::gxf::Expected<size_t> { HOLOSCAN_LOG_ERROR("Unable to serialize message"); return 0; }; inline static DeserializeFunc none_deserialize = []([[maybe_unused]] GXFEndpoint* buffer) -> nvidia::gxf::Expected<Message> { HOLOSCAN_LOG_ERROR("Unable to deserialize message"); return Message(); }; inline static Codec none_codec = std::make_pair(none_serialize, none_deserialize); static CodecRegistry& get_instance(); static nvidia::gxf::Expected<size_t> serialize(const Message& message, GXFEndpoint* gxf_endpoint) { auto& instance = get_instance(); const std::type_index index = std::type_index(message.value().type()); const SerializeFunc& func = instance.get_serializer(index); return func(message, gxf_endpoint); } template

<typename typeT> static nvidia::gxf::Expected<Message> deserialize(GXFEndpoint* gxf_endpoint) { auto& instance = get_instance(); const std::type_index index = std::type_index(typeid(typeT)); const DeserializeFunc& func = instance.get_deserializer(index); return func(gxf_endpoint); } Codec& get_codec(const std::type_index& index) { auto maybe_name = index_to_name(index); if (!maybe_name) { HOLOSCAN_LOG_WARN("No codec for type '{}' exists", index.name()); return CodecRegistry::none_codec; } auto& codec = codec_map_[maybe_name.value()]; return codec; } Codec& get_codec(const std::string& codec_name) { auto loc = codec_map_.find(codec_name); if (loc == codec_map_.end()) { HOLOSCAN_LOG_WARN("No codec for name '{}' exists", codec_name); return CodecRegistry::none_codec; } auto& codec = loc->second; return codec; } SerializeFunc& get_serializer(const std::string& codec_name) { auto loc = codec_map_.find(codec_name); if (loc == codec_map_.end()) { HOLOSCAN_LOG_WARN("No serializer for name '{}' exists", codec_name); return CodecRegistry::none_serialize; } Codec& codec = loc->second; return codec.first; } SerializeFunc& get_serializer(const std::type_index& index) { auto maybe_name = index_to_name(index); if (!maybe_name) { HOLOSCAN_LOG_WARN("No serializer for type '{}' exists", index.name()); return CodecRegistry::none_serialize; } auto& serializer = codec_map_[maybe_name.value()].first; return serializer; } DeserializeFunc& get_deserializer(const std::string& codec_name) { auto loc = codec_map_.find(codec_name); if (loc == codec_map_.end()) { HOLOSCAN_LOG_WARN("No deserializer for name '{}' exists", codec_name); return CodecRegistry::none_deserialize; } Codec& codec = loc->second; return codec.second; } DeserializeFunc& get_deserializer(const std::type_index& index) { auto maybe_name = index_to_name(index); if (!maybe_name) { HOLOSCAN_LOG_WARN("No deserializer for type '{}' exists", index.name()); return CodecRegistry::none_deserialize; } auto& deserializer = codec_map_[maybe_name.value()].second; return deserializer; } expected<std::type_index, RuntimeError> name_to_index(const std::string& codec_name) { auto loc = name_to_index_map_.find(codec_name); if (loc == name_to_index_map_.end()) { auto err_msg = fmt::format("No codec for name '{}' exists", codec_name); return make_unexpected<RuntimeError>(RuntimeError(ErrorCode::kCodecError, err_msg)); } return loc->second; } expected<std::string, RuntimeError> index_to_name(const std::type_index& index) { auto loc = index_to_name_map_.find(index); if (loc == index_to_name_map_.end()) { auto err_msg = fmt::format("No codec for type '{}' exists", index.name()); return

```
make_unexpected<RuntimeError>(RuntimeError(ErrorCode::kCodecError, err_msg));
} return loc->second; } template <typename typeT> void
add_codec(std::pair<SerializeFunc, DeserializeFunc> codec, const std::string&
codec_name, bool overwrite = true) { auto index = std::type_index(typeid(typeT));
add_codec(index, codec, codec_name, overwrite); } void add_codec(const
std::type_index& index, std::pair<SerializeFunc, DeserializeFunc> codec, const
std::string& codec_name, bool overwrite = true) { auto name_search =
name_to_index_map_.find(codec_name); if (name_search !=
name_to_index_map_.end()) { if (!overwrite) { HOLOSCAN_LOG_INFO("Existing codec
for name '{}' found.", codec_name); return; } if (index != name_search->second) {
HOLOSCAN_LOG_ERROR("Existing codec for name '{}' found, but with non-matching
type_index."); } HOLOSCAN_LOG_INFO("Replacing existing codec with name '{}'.",
codec_name); codec_map_.erase(codec_name); }
name_to_index_map_.try_emplace(codec_name, index);
index_to_name_map_.try_emplace(index, codec_name);
codec_map_.try_emplace(codec_name, codec); } template <typename typeT> void
add_codec(const std::string& codec_name, bool overwrite = true) { auto
name_search = name_to_index_map_.find(codec_name); auto index =
std::type_index(typeid(typeT)); if (name_search != name_to_index_map_.end()) { if
(!overwrite) { HOLOSCAN_LOG_INFO("Existing codec for name '{}' found.",
codec_name); return; } if (index != name_search->second) {
HOLOSCAN_LOG_ERROR("Existing codec for name '{}' found, but with non-matching
type_index."); } HOLOSCAN_LOG_INFO("Replacing existing codec with name '{}'.",
codec_name); codec_map_.erase(codec_name); }
name_to_index_map_.try_emplace(codec_name, index);
index_to_name_map_.try_emplace(index, codec_name); codec_map_.emplace(
codec_name, std::make_pair( [](const Message& data, GXFEndpoint* gxf_endpoint) -
> nvidia::gxf::Expected<size_t> { try { const typeT& value = std::any_cast<typeT>
(data.value()); Endpoint endpoint(gxf_endpoint); auto result =
codec<typeT>::serialize(value, &endpoint); if (result) { return result.value(); } else {
HOLOSCAN_LOG_ERROR("Error happens in serializing data of type '{}'",
typeid(typeT).name()); return nvidia::gxf::Unexpected(GXF_FAILURE); } } catch (const
std::bad_any_cast& e) { HOLOSCAN_LOG_ERROR("Unable to cast the data (std::any)
to '{}' : {}", typeid(typeT).name(), e.what()); return
nvidia::gxf::Unexpected(GXF_FAILURE); } }, [](GXFEndpoint* gxf_endpoint) ->
nvidia::gxf::Expected<Message> { Endpoint endpoint(gxf_endpoint); auto
```

maybe_value = codec<typeT>::deserialize(&endpoint); if (maybe_value) { return Message{maybe_value.value()}; } else { HOLOSCAN_LOG_ERROR("Error happens in deserializing data of type '{}'", typeid(typeT).name()); return nvidia::gxf::Unexpected(GXF_FAILURE); } })); } private: CodecRegistry() { *// Note: All codecs are for a holoscan::Message. // The names used here could be holoscan::Message(bool), etc. // For conciseness, I just used the message's internal value type for the name.* add_codec<bool>("bool"s); add_codec<int8_t>("int8_t"s); add_codec<int16_t>("int16_t"s); add_codec<int32_t>("int32_t"s); add_codec<int64_t>("int64_t"s); add_codec<uint8_t>("uint8_t"s); add_codec<uint16_t>("uint16_t"s); add_codec<uint32_t>("uint32_t"s); add_codec<uint64_t>("uint64_t"s); add_codec<float>("float"s); add_codec<double>("double"s); add_codec<std::complex<float>>("std::complex<float>"s); add_codec<std::complex<double>>("std::complex<double>"s); add_codec<std::string>("std::string"s); add_codec<std::vector<bool>>("std::vector<bool>"s); add_codec<std::vector<int8_t>>("std::vector<int8_t>"s); add_codec<std::vector<int16_t>>("std::vector<int16_t>"s); add_codec<std::vector<int32_t>>("std::vector<int32_t>"s); add_codec<std::vector<int64_t>>("std::vector<int64_t>"s); add_codec<std::vector<uint8_t>>("std::vector<uint8_t>"s); add_codec<std::vector<uint16_t>>("std::vector<uint16_t>"s); add_codec<std::vector<uint32_t>>("std::vector<uint32_t>"s); add_codec<std::vector<uint64_t>>("std::vector<uint64_t>"s); add_codec<std::vector<float>>("std::vector<float>"s); add_codec<std::vector<double>>("std::vector<double>"s); add_codec<std::vector<std::complex<float>>>("std::vector<std::complex<float>>"s); add_codec<std::vector<std::complex<double>>>("std::vector<std::complex<double>>"s); add_codec<std::vector<std::string>>("std::vector<std::string>"s); add_codec<std::vector<std::vector<bool>>>("std::vector<std::vector<bool>>"s); add_codec<std::vector<std::vector<int8_t>>>("std::vector<std::vector<int8_t>>"s); add_codec<std::vector<std::vector<int16_t>>>("std::vector<std::vector<int16_t>>"s); add_codec<std::vector<std::vector<int32_t>>>("std::vector<std::vector<int32_t>>"s); add_codec<std::vector<std::vector<int64_t>>>("std::vector<std::vector<int64_t>>"s); add_codec<std::vector<std::vector<uint8_t>>>

```
("std::vector<std::vector<uint8_t>>>"s);
add_codec<std::vector<std::vector<uint16_t>>>
("std::vector<std::vector<uint16_t>>>"s);
add_codec<std::vector<std::vector<uint32_t>>>
("std::vector<std::vector<uint32_t>>>"s);
add_codec<std::vector<std::vector<uint64_t>>>
("std::vector<std::vector<uint64_t>>>"s); add_codec<std::vector<std::vector<float>>>
("std::vector<std::vector<float>>>"s); add_codec<std::vector<std::vector<double>>>
("std::vector<std::vector<double>>>"s);
add_codec<std::vector<std::vector<std::complex<float>>>>(
"std::vector<std::vector<std::complex<float>>>>"s);
add_codec<std::vector<std::vector<std::complex<double>>>>(
"std::vector<std::vector<std::complex<double>>>>"s);
add_codec<std::vector<std::vector<std::string>>>
("std::vector<std::vector<std::string>>>"s); // shared_ptr variants for all of the above
add_codec<std::shared_ptr<bool>>("std::shared_ptr<bool>"s);
add_codec<std::shared_ptr<int8_t>>("std::shared_ptr<int8_t>"s);
add_codec<std::shared_ptr<int16_t>>("std::shared_ptr<int16_t>"s);
add_codec<std::shared_ptr<int32_t>>("std::shared_ptr<int32_t>"s);
add_codec<std::shared_ptr<int64_t>>("std::shared_ptr<int64_t>"s);
add_codec<std::shared_ptr<uint8_t>>("std::shared_ptr<uint8_t>"s);
add_codec<std::shared_ptr<uint16_t>>("std::shared_ptr<uint16_t>"s);
add_codec<std::shared_ptr<uint32_t>>("std::shared_ptr<uint32_t>"s);
add_codec<std::shared_ptr<uint64_t>>("std::shared_ptr<uint64_t>"s);
add_codec<std::shared_ptr<float>>("std::shared_ptr<float>"s);
add_codec<std::shared_ptr<double>>("std::shared_ptr<double>"s);
add_codec<std::shared_ptr<std::complex<float>>>
("std::shared_ptr<std::complex<float>>"s);
add_codec<std::shared_ptr<std::complex<double>>>
("std::shared_ptr<std::complex<double>>"s);
add_codec<std::shared_ptr<std::string>>("std::shared_ptr<std::string>"s);
add_codec<std::shared_ptr<std::vector<bool>>>
("std::shared_ptr<std::vector<bool>>"s);
add_codec<std::shared_ptr<std::vector<int8_t>>>
("std::shared_ptr<std::vector<int8_t>>"s);
add_codec<std::shared_ptr<std::vector<int16_t>>>
```

```
("std::shared_ptr<std::vector<int16_t>>"s);
add_codec<std::shared_ptr<std::vector<int32_t>>>
("std::shared_ptr<std::vector<int32_t>>"s);
add_codec<std::shared_ptr<std::vector<int64_t>>>
("std::shared_ptr<std::vector<int64_t>>"s);
add_codec<std::shared_ptr<std::vector<uint8_t>>>
("std::shared_ptr<std::vector<uint8_t>>"s);
add_codec<std::shared_ptr<std::vector<uint16_t>>>
("std::shared_ptr<std::vector<uint16_t>>"s);
add_codec<std::shared_ptr<std::vector<uint32_t>>>
("std::shared_ptr<std::vector<uint32_t>>"s);
add_codec<std::shared_ptr<std::vector<uint64_t>>>
("std::shared_ptr<std::vector<uint64_t>>"s);
add_codec<std::shared_ptr<std::vector<float>>>
("std::shared_ptr<std::vector<float>>"s);
add_codec<std::shared_ptr<std::vector<double>>>
("std::shared_ptr<std::vector<double>>"s);
add_codec<std::shared_ptr<std::vector<std::complex<float>>>>(
"std::shared_ptr<std::vector<std::complex<float>>>"s);
add_codec<std::shared_ptr<std::vector<std::complex<double>>>>(
"std::shared_ptr<std::vector<std::complex<double>>>"s);
add_codec<std::shared_ptr<std::vector<std::string>>>(
"std::shared_ptr<std::vector<std::string>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<bool>>>>(
"std::shared_ptr<std::vector<std::vector<bool>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<int8_t>>>>(
"std::shared_ptr<std::vector<std::vector<int8_t>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<int16_t>>>>(
"std::shared_ptr<std::vector<std::vector<int16_t>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<int32_t>>>>(
"std::shared_ptr<std::vector<std::vector<int32_t>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<int64_t>>>>(
"std::shared_ptr<std::vector<std::vector<int64_t>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<uint8_t>>>>(
"std::shared_ptr<std::vector<std::vector<uint8_t>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<uint16_t>>>>(
```

```
"std::shared_ptr<std::vector<std::vector<uint16_t>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<uint32_t>>>>(
"std::shared_ptr<std::vector<std::vector<uint32_t>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<uint64_t>>>>(
"std::shared_ptr<std::vector<std::vector<uint64_t>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<float>>>>(
"std::shared_ptr<std::vector<std::vector<float>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<double>>>>(
"std::shared_ptr<std::vector<std::vector<double>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<std::complex<float>>>>>(
"std::shared_ptr<std::vector<std::vector<std::complex<float>>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<std::complex<double>>>>>(
"std::shared_ptr<std::vector<std::vector<std::complex<double>>>>>"s);
add_codec<std::shared_ptr<std::vector<std::vector<std::string>>>>(
"std::shared_ptr<std::vector<std::vector<std::string>>>>"s); // add code for the
camera pose array used by HolovizOp add_codec<std::shared_ptr<std::array<float,
16>>>("std::shared_ptr<std::array<float, 16>>"s); } // define maps to and from
type_index and string (since type_index may vary across platforms)
std::unordered_map<std::type_index, std::string> index_to_name_map_;
std::unordered_map<std::string, std::type_index> name_to_index_map_;
std::unordered_map<std::string, std::pair<SerializeFunc, DeserializeFunc>>
codec_map_; }; } // namespace holoscan #endif/*
HOLOSCAN_CORE_CODEC_REGISTRY_HPP */
```