D:\MAL23-P7-G5-Main\Engine\Source\Utilities\CommonUtilities\include\CommonUtilities\Structures\Blackboard.hpp

```
#pragma once

#include <string>
#include <unordered_map>
#include <memory>
#include <cassert>
#include <shared_mutex>

#include <CommonUtilities/Config.h>
#include <CommonUtilities/System/IDGenerator.h>
#include <CommonUtilities/Structures/FreeVector.hpp>
#include <CommonUtilities/Utility/JsonUtils.hpp>
#include <CommonUtilities/Utility/Concepts.hpp>
#include <CommonUtilities/Utility/Clonable.hpp>

namespace CommonUtilities
{
    template<typename IDType = std::string_view, typename Hash = std::hash<IDType>> requires IsHashableType<Hash, IDType>
    class Blackboard
    {
    public:
        Blackboard() = default;
        ~Blackboard() = default;

        Blackboard(Blackboard&& aOther);
        Blackboard(const Blackboard& aOther);

        Blackboard& operator=(Blackboard&& aOther);
        Blackboard& operator=(const Blackboard& aOther);

        template<typename T>
        NODISC const T& Get(const IDType& aID) const;

        template<typename T>
```

```
35          NODISC T& Get(const IDType& aID);
36
37          template<typename T>
38          NODISC const T* TryGet(const IDType& aID) const;
39
40          template<typename T>
41          NODISC T* TryGet(const IDType& aID);
42
43          template<typename T>
44          void Set(const IDType& aID, T&& aValue);
45
46          template<typename T, typename... Args> requires std::constructible_from<T, Args...>
47          void Emplace(const IDType& aID, Args&&... someArgs)
48          {
49              using ValueType = std::decay_t<T>;
50
51              std::scoped_lock lock(myMutex);
52
53              ValueMap<ValueType>& map = FindValueMap<ValueType>();
54              map.Emplace(aID, std::forward<Args>(someArgs)...);
55          }
56
57          template<typename T>
58          void Erase(const IDType& aID);
59
60          template<typename T>
61          NODISC bool Has(const IDType& aID) const;
62
63          template<typename T>
64          NODISC bool HasType() const;
65
66          void EraseKey(const IDType& aID);
67
68          void Clear();
69
70      private:
71          class ValueMapBase
```

```
        {
        public:
            ValueMapBase() = default;
            virtual ~ValueMapBase() = default;

            NODISC virtual bool Has(const IDType& aID) = 0;

            virtual void Erase(const IDType& aID) = 0;
            virtual void Clear() = 0;

            NODISC virtual std::unique_ptr<ValueMapBase> Clone() const = 0;

        private:
            bool dummy = false;
            NLOHMANN_DEFINE_TYPE_INTRUSIVE(ValueMapBase, dummy);
        };

        template<typename T>
        class ValueMap final : public Clonable<ValueMapBase, ValueMap<T>>
        {
        public:
            ValueMap() = default;
            ~ValueMap() = default;

            NODISC const T& Get(const IDType& aID) const
            {
                return myValues.at(myIndices.at(Hash{}(aID)));
            }
            NODISC T& Get(const IDType& aID)
            {
                return myValues.at(myIndices.at(Hash{}(aID)));
            }

            NODISC const T* TryGet(const IDType& aID) const
            {
                if (auto it = myIndices.find(Hash{}(aID)); it != myIndices.end())
                {
```

```
109                     return std::addressof(myValues.at(it->second));
110                 }
111 
112                 return nullptr;
113             }
114             NODISC T* TryGet(const IDType& aID)
115             {
116                 return const_cast<T*>(std::as_const(*this).TryGet(aID));
117             }
118 
119             template<typename... Args>
120             void Emplace(const IDType& aID, Args&&... someArgs)
121             {
122                 const std::size_t hash = Hash{}(aID);
123 
124                 if (const auto it = myIndices.find(hash); it != myIndices.end())
125                 {
126                     myValues[it->second] = T{ std::forward<Args>(someArgs)... };
127                 }
128                 else
129                 {
130                     std::size_t index = myValues.emplace(std::forward<Args>(someArgs)...);
131                     UNSD auto insert = myIndices.try_emplace(hash, index);
132                     assert(insert.second);
133                 }
134             }
135 
136             void Insert(const IDType& aID, const T& aValue)
137             {
138                 Emplace(aID, aValue);
139             }
140             void Insert(const IDType& aID, T&& aValue)
141             {
142                 Emplace(aID, std::move(aValue));
143             }
144 
145             NODISC bool Has(const IDType& aID) override
```

```
146             {
147                 return myIndices.find(Hash{}(aID)) != myIndices.end();
148             }
149 
150             void Erase(const IDType& aID) override
151             {
152                 if (const auto it = myIndices.find(Hash{}(aID)); it != myIndices.end())
153                 {
154                     myValues.erase(it->second);
155                     myIndices.erase(it);
156                 }
157             }
158             void Clear() override
159             {
160                 myValues.clear();
161                 myIndices.clear();
162             }
163 
164         private:
165             using IDIndicesMap = std::unordered_map<std::size_t, std::size_t>;
166 
167             FreeVector<T>   myValues;
168             IDIndicesMap    myIndices;
169 
170             friend void to_json(nlohmann::json& aJSON, const ValueMap& aValues)
171             {
172                 if constexpr (HasToJson<T>)
173                 {
174                     if constexpr (HasKeyType<T> && HasMappedType<T>)
175                     {
176                         if constexpr (HasToJson<typename T::key_type> && HasToJson<typename T::mapped_type>)
177                         {
178                             aJSON["myValues"] = aValues.myValues;
179                             aJSON["myIndices"] = aValues.myIndices;
180                         }
181                     }
182                     else if constexpr (HasValueType<T>)
```

```
183                     {
184                         if constexpr (HasToJson<typename T::value_type>)
185                         {
186                             aJSON["myValues"] = aValues.myValues;
187                             aJSON["myIndices"] = aValues.myIndices;
188                         }
189                     }
190                     else
191                     {
192                         aJSON["myValues"] = aValues.myValues;
193                         aJSON["myIndices"] = aValues.myIndices;
194                     }
195                 }
196             }
197             friend void from_json(const nlohmann::json& aJSON, ValueMap& aValues)
198             {
199                 if constexpr (HasFromJson<T>)
200                 {
201                     if constexpr (HasKeyType<T> && HasMappedType<T>)
202                     {
203                         if constexpr (HasFromJson<typename T::key_type> && HasFromJson<typename T::mapped_type>)
204                         {
205                             aValues.myValues = aJSON["myValues"];
206                             aValues.myIndices = aJSON["myIndices"];
207                         }
208                     }
209                     else if constexpr (HasValueType<T>)
210                     {
211                         if constexpr (HasFromJson<typename T::value_type>)
212                         {
213                             aValues.myValues = aJSON["myValues"];
214                             aValues.myIndices = aJSON["myIndices"];
215                         }
216                     }
217                     else
218                     {
219                         aValues.myValues = aJSON["myValues"];
```

```
                        aValues.myIndices = aJSON["myIndices"];
                    }
                }
            }
        };

        template<typename T>
        NODISC auto FindValueMap() const -> const ValueMap<T>&;

        template<typename T>
        NODISC auto FindValueMap() -> ValueMap<T>&;

        using TypeValueMap = std::unordered_map<std::size_t, std::unique_ptr<ValueMapBase>>;

        TypeValueMap myData;
        mutable std::shared_mutex myMutex;

        friend struct nlohmann::adl_serializer<std::unique_ptr<ValueMapBase>>;

        NLOHMANN_DEFINE_TYPE_INTRUSIVE(Blackboard, myData);
    };

    template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
    inline Blackboard<IDType, Hash>::Blackboard(Blackboard&& aOther)
    {
        std::scoped_lock lock(aOther.myMutex);
        myData = std::move(aOther.myData);
    }
    template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
    inline Blackboard<IDType, Hash>::Blackboard(const Blackboard& aOther)
    {
        std::shared_lock lock(aOther.myMutex);

        for (const auto& [id, map] : aOther.myData)
        {
            myData[id] = map->Clone();
        }
```

```
    }

    template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
    inline Blackboard<IDType, Hash>& Blackboard<IDType, Hash>::operator=(Blackboard&& aOther)
    {
        if (this == &aOther)
            return *this;

        std::scoped_lock lock1(myMutex);
        std::scoped_lock lock2(aOther.myMutex);

        myData = std::move(aOther.myData);

        return *this;
    }
    template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
    inline Blackboard<IDType, Hash>& Blackboard<IDType, Hash>::operator=(const Blackboard& aOther)
    {
        if (this == &aOther)
            return *this;

        std::scoped_lock lock1(myMutex);
        std::shared_lock lock2(aOther.myMutex);

        for (const auto& [id, map] : aOther.myData)
        {
            myData[id] = map->Clone();
        }

        return *this;
    }

    template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
    template<typename T>
    inline const T& Blackboard<IDType, Hash>::Get(const IDType& aID) const
    {
        std::shared_lock lock(myMutex);
```

```
294
295         const ValueMap<T>& map = FindValueMap<T>();
296         return map.Get(aID);
297     }
298
299     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
300     template<typename T>
301     inline T& Blackboard<IDType, Hash>::Get(const IDType& aID)
302     {
303         std::scoped_lock lock(myMutex);
304
305         ValueMap<T>& map = FindValueMap<T>();
306         return map.Get(aID);
307     }
308
309     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
310     template<typename T>
311     inline const T* Blackboard<IDType, Hash>::TryGet(const IDType& aID) const
312     {
313         std::scoped_lock lock(myMutex);
314
315         const ValueMap<T>& map = FindValueMap<T>();
316         return map.TryGet(aID);
317     }
318
319     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
320     template<typename T>
321     inline T* Blackboard<IDType, Hash>::TryGet(const IDType& aID)
322     {
323         std::scoped_lock lock(myMutex);
324
325         ValueMap<T>& map = FindValueMap<T>();
326         return map.TryGet(aID);
327     }
328
329     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
330     template<typename T>
```

```
331     inline void Blackboard<IDType, Hash>::Set(const IDType& aID, T&& aValue)
332     {
333         Emplace<T>(aID, std::forward<T>(aValue));
334     }
335 
336     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
337     template<typename T>
338     inline void Blackboard<IDType, Hash>::Erase(const IDType& aID)
339     {
340         std::scoped_lock lock(myMutex);
341 
342         ValueMap<T>& map = FindValueMap<T>();
343         map.Erase(aID);
344     }
345 
346     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
347     template<typename T>
348     inline bool Blackboard<IDType, Hash>::Has(const IDType& aID) const
349     {
350         std::shared_lock lock(myMutex);
351 
352         const ValueMap<T>& map = FindValueMap<T>();
353         return map.Has(aID);
354     }
355 
356     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
357     template<typename T>
358     inline bool Blackboard<IDType, Hash>::HasType() const
359     {
360         static constexpr std::size_t key = Type<T>::ID();
361 
362         const auto it = myData.find(key);
363         return it != myData.end();
364     }
365 
366     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
367     inline void Blackboard<IDType, Hash>::EraseKey(const IDType& aID)
```

```
368     {
369         std::scoped_lock lock(myMutex);
370 
371         for (auto& [id, map] : myData)
372         {
373             map.Erase(aID);
374         }
375     }
376 
377     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
378     inline void Blackboard<IDType, Hash>::Clear()
379     {
380         std::scoped_lock lock(myMutex);
381 
382         for (auto& [id, map] : myData)
383         {
384             map.Clear();
385         }
386     }
387 
388     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
389     template<typename T>
390     inline auto Blackboard<IDType, Hash>::FindValueMap() const -> const ValueMap<T>&
391     {
392         static constexpr std::size_t key = Type<T>::ID();
393         return *static_cast<ValueMap<T>*>(myData.at(key).get());
394     }
395 
396     template<typename IDType, typename Hash> requires IsHashableType<Hash, IDType>
397     template<typename T>
398     inline auto Blackboard<IDType, Hash>::FindValueMap() -> ValueMap<T>&
399     {
400         static constexpr std::size_t key = Type<T>::ID();
401 
402         if (const auto it = myData.find(key); it != myData.end())
403         {
404             return *static_cast<ValueMap<T>*>(it->second.get());
```

```
405         }
406
407         PolymorphicJsonSerializer<std::unique_ptr<ValueMapBase>>::template RegisterTypes<ValueMap<T>>();
408
409         UNSD auto insert = myData.try_emplace(key, std::make_unique<ValueMap<T>>());
410         assert(insert.second);
411
412         return *static_cast<ValueMap<T>*>(insert.first->second.get());
413     }
414 }
415
416 NLOHMANN_JSON_NAMESPACE_BEGIN
417
418 template<>
419 struct adl_serializer<std::unique_ptr<typename CommonUtilities::Blackboard<>::ValueMapBase>>
420     : cu::PolymorphicJsonSerializer<std::unique_ptr<typename CommonUtilities::Blackboard<>::ValueMapBase>>
421 {
422
423 };
424
425 NLOHMANN_JSON_NAMESPACE_END
```