Armyengine

1

Generated by Doxygen 1.8.3.1

Sat Apr 27 2013 16:52:03

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

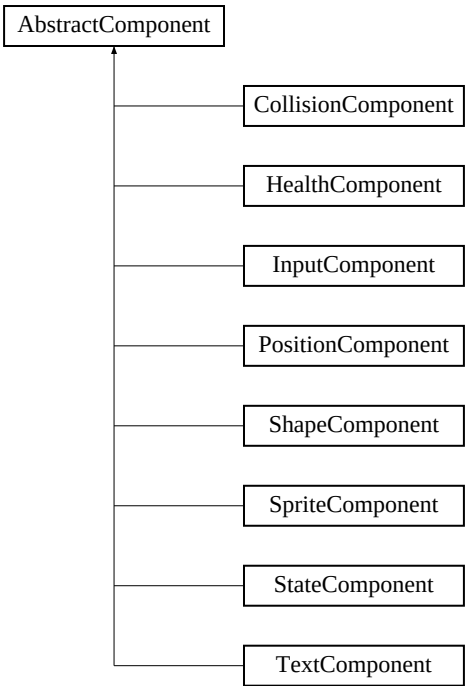
Class Documentation

4.1 AbstractComponent Class Reference

The abstract representation for each component.

`#include <AbstractComponent.h>`

Inheritance diagram for AbstractComponent:



Public Member Functions

- AbstractComponent (std::string, std::string)

  Constructor for abstract component.
- ~AbstractComponent ()

  Destructor for abstract component.
- const std::string & getName ()

  Returns the name of the component.
- const std::string & getFamily ()

Returns the family of the component.

- bool hasAttribute (const std::string &attr_key)

  Checks the attributeList for the given attribute.
- attribute_type getAttributeType (const std::string &attr_key)

  Gets the attribute type.
- int getAttribute_int (const std::string &attr_key)

  Get the attribute in the form of an integer.
- void setAttribute_int (const std::string &attr_key, int ivalue)

  Set the attribute for the given key with the given integer value.
- float getAttribute_float (const std::string &attr_key)

  Get the attribute in the form of an float.
- void setAttribute_float (const std::string &attr_key, float fvalue)

  Set the attribute for the given key with the given float value.
- const std::string & getAttribute_string (const std::string &attr_key)

  Get the attribute in the form of an string.
- void setAttribute_string (const std::string &attr_key, std::string svalue)

  Set the attribute for the given key with the given string value.
- componentFloatArrayType * getAttribute_floatArray (const std::string &attr_key)

  Get the attribute in the form of an float array.
- void setAttribute_floatArray (const std::string &attr_key, componentFloatArrayType sfvalue)

  Set the attribute for the given key with the given float array value.
- componentIntegerArrayType * getAttribute_intArray (std::string attr_key)

  Get the attribute in the form of an integer array.
- void setAttribute_intArray (const std::string &attr_key, componentIntegerArrayType sivalue)

  Set the attribute for the given key with the given integer array value.
- attribute getAttribute (const std::string &attr_key)

  Gets the attribute structure.
- void **setAttribute** (const std::string &, int)
- void **setAttribute** (const std::string &, float)
- void **setAttribute** (const std::string &, std::string)
- void **setAttribute** (const std::string &, componentFloatArrayType)
- void **setAttribute** (const std::string &, componentIntegerArrayType)
- virtual int update ()=0

  Updates the component based on the current set of attributes.

4.1.1 Detailed Description

The abstract representation for each component.

The abstract component contains methods to add, remove and modify attributes within the component. The family member is overidden to describe the component type, and the name member is a unique name assigned to the given component.

4.1.2 Member Function Documentation

4.1.2.1 attribute AbstractComponent::getAttribute (const std::string & *attr_key*)

Gets the attribute structure.

Gets the attribute by the given key name.

Parameters

attr_key	The key name for the attribute

Returns

The attribute structure.

```
140                                                                 {
141     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
142     return this->attributeList[attr_key];
143 }
```

4.1.2.2 float AbstractComponent::getAttribute_float (const std::string & *attr_key*)

Get the attribute in the form of an float.

Gets the attribute with the given keyname, and returns the value stored for that attribute. If the attribute has a different type than described by the attribute, you will get unwanted behaviour.

Parameters

attr_key	The key name for the attribute

Returns

The value for the given attribute.

```
62                                                                  {
63     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
64     assert(this->attributeList[attr_key].attr_type == ATTR_FLOAT && "attribute not of type float");
65     return this->attributeList[attr_key].attr_container.f;
66 }
```

4.1.2.3 componentFloatArrayType * AbstractComponent::getAttribute_floatArray (const std::string & *attr_key*)

Get the attribute in the form of an float array.

Gets the attribute with the given keyname, and returns the value stored for that attribute. If the attribute has a different type than described by the attribute, you will get unwanted behaviour.

Parameters

attr_key	The key name for the attribute

Returns

The value for the given attribute.

```
100                                                                                 {
101     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
102     assert(this->attributeList[attr_key].attr_type == ATTR_FLOATARRAY && "attribute not of type float array
    ");
103     return this->attributeList[attr_key].attr_container.sf;
104 }
```

4.1.2.4 int AbstractComponent::getAttribute_int (const std::string & *attr_key*)

Get the attribute in the form of an integer.

Gets the attribute with the given keyname, and returns the value stored for that attribute. If the attribute has a different type than described by the attribute, you will get unwanted behaviour.

Parameters

attr_key	The key name for the attribute

Returns

The value for the given attribute.

```
43                                                                  {
44     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
45     assert(this->attributeList[attr_key].attr_type == ATTR_INTEGER && "attribute not of type integer");
46     return this->attributeList[attr_key].attr_container.i;
47 }
```

4.1.2.5 componentIntegerArrayType * AbstractComponent::getAttribute_intArray (std::string *attr_key*)

Get the attribute in the form of an integer array.

Gets the attribute with the given keyname, and returns the value stored for that attribute. If the attribute has a different type than described by the attribute, you will get unwanted behaviour.

Parameters

attr_key	The key name for the attribute

Returns

The value for the given attribute.

```
120                                                                                                 {
121     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
122     assert(this->attributeList[attr_key].attr_type == ATTR_INTEGERARRAY && "attribute not of type int array
      ");
123     return this->attributeList[attr_key].attr_container.si;
124 }
```

4.1.2.6 const std::string & AbstractComponent::getAttribute_string (const std::string & *attr_key*)

Get the attribute in the form of an string.

Gets the attribute with the given keyname, and returns the value stored for that attribute. If the attribute has a different type than described by the attribute, you will get unwanted behaviour.

Parameters

attr_key	The key name for the attribute

Returns

The value for the given attribute.

```
81                                                                      {
82     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
83     assert(this->attributeList[attr_key].attr_type == ATTR_STRING && "attribute not of type string");
84     return *(this->attributeList[attr_key].attr_container.s);
85 }
```

4.1.2.7 attribute_type AbstractComponent::getAttributeType (const std::string & *attr_key*)

Gets the attribute type.

Returns the enum value for the given attribute key

Parameters

attr_key	The key name for the attribute.

Returns

The type of the attribute

```
38                                                                  {
39     assert(this->hasAttribute(attr_key) && "attr_key does not exist");
40     return this->attributeList[attr_key].attr_type;
41 }
```

4.1.2.8 const std::string & AbstractComponent::getName ( )

Returns the name of the component.

Returns

The name of the component

```
23                                          {
24     return this->name;
25 }
```

4.1.2.9 bool AbstractComponent::hasAttribute (const std::string & *attr_key*)

Checks the attributeList for the given attribute.

The attribute key name is passed, and the attribute list is searched. The function returns a boolean on whether or not it was successful in finding that attribute by the given key.

Parameters

attr_key	The key name of the attribute.

Returns

A boolean, where *true* means that the given attribute exists.

```
31                                                          {
32     if (this->attributeList.find(attr_key) == this->attributeList.end()) {
33         return false;
34     }
35     return true;
36 }
```

4.1.2.10 void AbstractComponent::setAttribute_float (const std::string & *attr_key*, float *fvalue*)

Set the attribute for the given key with the given float value.

The attribute key provided is checked for its existence. If it exists, the attribute is modified to be of the float type, and the data is modified to be of the float value.

If the attribute doesn't already exist, the attribute is created with the given key and value and placed within the attributeList.

Parameters

attr_key	The key name for the attribute
fvalue	The float value to set the given attribute to.

```
68                                                                                  {
69     if (this->hasAttribute(attr_key)) {
70         this->attributeList[attr_key].attr_container.f = fvalue;
71         this->attributeList[attr_key].attr_type = ATTR_FLOAT;
72     }
73     else {
74         attribute attr;
75         attr.attr_container.f = fvalue;
76         attr.attr_type = ATTR_FLOAT;
77         this->attributeList[attr_key] = attr;
78     }
79 }
```

4.1.2.11 void AbstractComponent::setAttribute_floatArray (const std::string & *attr_key*, componentFloatArrayType *sfvalue*)

Set the attribute for the given key with the given float array value.

The attribute key provided is checked for its existence. If it exists, the attribute is modified to be of the float array type, and the data is modified to be of the float array value.

If the attribute doesn't already exist, the attribute is created with the given key and value and placed within the attributeList.

Parameters

attr_key	The key name for the attribute
sfvalue	The float array value to set the given attribute to.

```
106                                                                                 {
107     if (this->hasAttribute(attr_key)) {
108         delete this->attributeList[attr_key].attr_container.sf;
109         this->attributeList[attr_key].attr_container.sf = new
      componentFloatArrayType(sfvalue);
110         this->attributeList[attr_key].attr_type = ATTR_FLOATARRAY;
111     }
112     else {
113         attribute attr;
114         attr.attr_container.sf = new componentFloatArrayType(sfvalue);
115         attr.attr_type = ATTR_FLOATARRAY;
116         this->attributeList[attr_key] = attr;
117     }
118 }
```

4.1.2.12 void AbstractComponent::setAttribute_int (const std::string & *attr_key*, int *ivalue*)

Set the attribute for the given key with the given integer value.

The attribute key provided is checked for its existence. If it exists, the attribute is modified to be of the integer type, and the data is modified to be of the integer value.

If the attribute doesn't already exist, the attribute is created with the given key and value and placed within the attributeList.

Parameters

attr_key	The key name for the attribute
ivalue	The integer value to set the given attribute to.

```
49                                                                                  {
50     if (this->hasAttribute(attr_key)) {
51         this->attributeList[attr_key].attr_container.i = ivalue;
52         this->attributeList[attr_key].attr_type = ATTR_INTEGER;
53     }
54     else {
55         attribute attr;
56         attr.attr_container.i = ivalue;
57         attr.attr_type = ATTR_INTEGER;
58         this->attributeList[attr_key] = attr;
59     }
```

```
60 }
```

4.1.2.13 void AbstractComponent::setAttribute_intArray (const std::string & *attr_key*, componentIntegerArrayType *sivalue*)

Set the attribute for the given key with the given integer array value.

The attribute key provided is checked for its existence. If it exists, the attribute is modified to be of the integer array type, and the data is modified to be of the integer array value.

If the attribute doesn't already exist, the attribute is created with the given key and value and placed within the attributeList.

Parameters

attr_key	The key name for the attribute
sivalue	The integer array value to set the given attribute to.

```
126                                                                                                                 {
127     if (this->hasAttribute(attr_key)) {
128         delete this->attributeList[attr_key].attr_container.si;
129         this->attributeList[attr_key].attr_container.si = new
    componentIntegerArrayType(sivalue);
130         this->attributeList[attr_key].attr_type = ATTR_INTEGERARRAY;
131     }
132     else {
133         attribute attr;
134         attr.attr_container.si = new componentIntegerArrayType(sivalue);
135         attr.attr_type = ATTR_INTEGERARRAY;
136         this->attributeList[attr_key] = attr;
137     }
138 }
```

4.1.2.14 void AbstractComponent::setAttribute_string (const std::string & *attr_key*, std::string *svalue*)

Set the attribute for the given key with the given string value.

The attribute key provided is checked for its existence. If it exists, the attribute is modified to be of the string type, and the data is modified to be of the string value.

If the attribute doesn't already exist, the attribute is created with the given key and value and placed within the attributeList.

Parameters

attr_key	The key name for the attribute
svalue	The string value to set the given attribute to.

```
87                                                                                                                  {
88     if (this->hasAttribute(attr_key)) {
89         *(this->attributeList[attr_key].attr_container.s) = svalue;
90         this->attributeList[attr_key].attr_type = ATTR_STRING;
91     }
92     else {
93         attribute attr;
94         attr.attr_container.s = new std::string(svalue);
95         attr.attr_type = ATTR_STRING;
96         this->attributeList[attr_key] = attr;
97     }
98 }
```

4.1.2.15 virtual int AbstractComponent::update ( ) [pure virtual]

Updates the component based on the current set of attributes.

Used to update the component. This should be performed after modifications have been made on the component's attributes or members.

Returns

Returns a non-zero value if it is successful.

Implemented in ShapeComponent, TextComponent, CollisionComponent, StateComponent, PositionComponent, InputComponent, SpriteComponent, and HealthComponent.

The documentation for this class was generated from the following files:

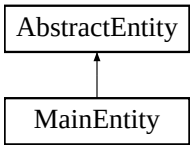
- AbstractComponent.h
- AbstractComponent.cpp

4.2 AbstractEntity Class Reference

The abstract representation for each entity.

`#include <AbstractEntity.h>`

Inheritance diagram for AbstractEntity:



Public Member Functions

- AbstractEntity (const std::string &, const std::string &, const int)
  AbstractEntity Constructor.
- const std::string & getName ()
  Getter for the name.
- const std::string & getFamily ()
  Getter for the family.
- int getID ()
  Getter for the ID.
- bool hasComponentName (const std::string &name)
  Check to see if a given component exists with the given name.
- bool hasComponentFamily (const std::string &family)
  Check to see if any component exists with the given family.
- void addComponent (std::shared_ptr< componentType > component)
  Add a component to given entity.
- const componentVectorType & getAllComponents ()
  Grab all of the components currently within the entity.
- std::list< std::shared_ptr < componentType > > getComponentsByFamily (const std::string &family)
  Returns a list of components which belongs to the given family.
- std::shared_ptr< componentType > getComponentByName (const std::string &name)
  Returns a component by its given name.
- virtual int handle ()=0
  virtual function to perform pre-initialization and handling

4.2.1 Detailed Description

The abstract representation for each entity.

The abstract entity contains methods to add, delete, and retrieve components, as well as a plethora of methods to find components by a given family, or by a given name.

Each entity should have a unique assigned to it, which is preserved by using the ArmyEngine::EntityFactory.

4.2.2 Member Function Documentation

4.2.2.1 void AbstractEntity::addComponent (std::shared_ptr< componentType > *component*)

Add a component to given entity.

Parameters

component	Is the shared_ptr of the component you wish to add the to the given entity.

```
55                                                                          {
56     assert(!this->hasComponentName(component->getName()) && "Already has component by that
       name");
57     this->componentVector.push_back(component);
58     return;
59 }
```

4.2.2.2 const componentVectorType & AbstractEntity::getAllComponents ( )

Grab all of the components currently within the entity.

Returns

The list of all components within the entity.

```
61                                                  {
62     return this->componentVector;
63 }
```

4.2.2.3 std::shared_ptr< componentType > AbstractEntity::getComponentByName (const std::string & *name*)

Returns a component by its given name.

Parameters

name	of the component

Returns

the component, otherwise it returns shared_ptr(nullptr)

```
75                                                                                  {
76     assert(this->hasComponentName(name) && "No component by that name");
77     auto iter = std::find_if(this->componentVector.begin(), this->componentVector.end(),
78         [&name] (std::shared_ptr<componentType> elem) {
79             if (elem->getName() == name) {
80                 return true;
81             }
82             return false;
83     });
84     if (iter != this->componentVector.end()) {
85         return *iter;
86     }
87     return std::shared_ptr<componentType> (nullptr);
88 }
```

4.2.2.4 std::list< std::shared_ptr< componentType > > AbstractEntity::getComponentsByFamily (const std::string & *family*)

Returns a list of components which belongs to the given family.

Parameters

family	Is the family of components that you wish to return.

Returns

The list of all components with the given family.

```
65                                                                                                                  {
66      std::list<std::shared_ptr<componentType>> componentList;
67      for(auto component : this->componentVector) {
68          if (family == component->getFamily()) {
69              componentList.push_back(component);
70          }
71      }
72      return componentList;
73  }
```

4.2.2.5 const std::string & AbstractEntity::getFamily ( )

Getter for the family.

Returns

The family of the entity

```
23                                          {
24      return this->family;
25  }
```

4.2.2.6 int AbstractEntity::getID ( )

Getter for the ID.

Returns

The unique ID of the entity

```
15                          {
16      return this->ID;
17  }
```

4.2.2.7 const std::string & AbstractEntity::getName ( )

Getter for the name.

Returns

The name of the entity

```
19                                          {
20      return this->name;
21  }
```

4.2.2.8 virtual int AbstractEntity::handle ( ) [pure virtual]

virtual function to perform pre-initialization and handling

Currently it is being used to sort the shape, text, and sprite components to provide Z-buffer ordering. It is advised that this method be used if any components are added during execution to prevent erroneous behaviour.

Returns

A non-zero value when it is successful.

Implemented in MainEntity.

4.2.2.9 bool AbstractEntity::hasComponentFamily (const std::string & *family*)

Check to see if any component exists with the given family.

Parameters

family	Is the family of the component you wish to find.

Returns

Returns true if a component exists with the given family, otherwise it returns false.

```
41                                                          {
42     auto iter = std::find_if(this->componentVector.begin(), this->componentVector.end(),
43         [&family] (std::shared_ptr<componentType> elem) {
44             if (elem->getFamily() == family) {
45                 return true;
46             }
47             return false;
48     });
49     if (iter != this->componentVector.end()) {
50         return true;
51     }
52     return false;
53 }
```

4.2.2.10 bool AbstractEntity::hasComponentName (const std::string & *name*)

Check to see if a given component exists with the given name.

Parameters

name	Is the unique name of the component you wish to find.

Returns

Returns true if a component exists with the given name, otherwise it returns false.

```
27                                                          {
28     auto iter = std::find_if(this->componentVector.begin(), this->componentVector.end(),
29         [&name] (std::shared_ptr<componentType> elem) {
30             if (elem->getName() == name) {
31                 return true;
32             }
33             return false;
34     });
35     if (iter != componentVector.end()) {
36         return true;
37     }
38     return false;
39 }
```

The documentation for this class was generated from the following files:

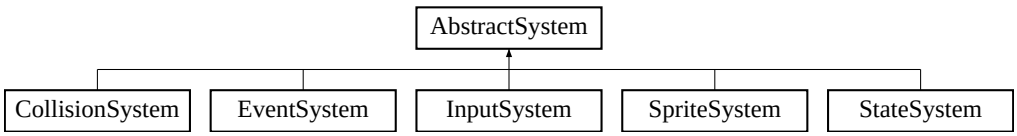
- AbstractEntity.h
- AbstractEntity.cpp

4.3 AbstractSystem Class Reference

The abstract representation for each system.

`#include <AbstractSystem.h>`

Inheritance diagram for AbstractSystem:



Public Member Functions

- AbstractSystem (std::string name)

  Abstract system constructor.
- ~AbstractSystem ()

  Abstract system destructor.
- std::string getName ()

  Gets the name of the current system.
- virtual int process ()=0

  Perform actions on entities.

Protected Attributes

- EntityManager * entityManager

  a pointer to the entity manager singleton

4.3.1 Detailed Description

The abstract representation for each system.

The abstract system has methods to get the name of the system, and to process entities stored within the entity manager.

Systems carry out specific needs for the engine, such as rendering stuff onto the screen, detecting if two entities have collided, or checking to see if certain events have been performed, and reacting to it.

4.3.2 Member Function Documentation

4.3.2.1 std::string AbstractSystem::getName ( )

Gets the name of the current system.

Returns

The name of the current system

```
14                                  {
15     return this->name;
16 }
```

4.3.2.2 virtual int AbstractSystem::process ( ) [pure virtual]

Perform actions on entities.

The process method is a pure virtual function that processes all of th entities stored within the entity manager. Some systems do not use the entity manager for processing, but may process other things that require processing per frame.

What is processed is fully dependant on the implementation.

Returns

A non-zero value if the processing was successful.

Implemented in CollisionSystem, EventSystem, StateSystem, InputSystem, and SpriteSystem.

The documentation for this class was generated from the following files:

- AbstractSystem.h
- AbstractSystem.cpp

4.4 ArmyEngine Class Reference

Army engine singleton facade. Frontend to the engine.

`#include <ArmyEngine.h>`

Public Member Functions

- ComponentFactory & getComponentFactory ()

  Used to get the component factory instance.
- EntityFactory & getEntityFactory ()

  Used to get the entity factory instance.
- void setStateMachineFunction (functionStateMachineTemplate func)

  Used to set the state machine function.
- void addEntity (std::shared_ptr< entityType >)

  Add an entity to the engine.
- void addCollisionCallback (collisionTagTuple, functionCollisionTemplate)

  Add a collision callback to the engine.
- void addStateCallback (std::string, functionTemplate)

  Add a state callback to the engine.
- void addEventCallback (EnumEventType, functionRegisterTemplate)

  Add an event callback to the engine.
- void addInputCallback (std::string, functionEventTemplate)

  Add an input callback to the engine.
- sf::RenderWindow * getWindow ()

  Get the instance of the application window.

- void setWindowSize (int width, int height)

  Set the window size (TODO: fix this)
- void setStateEnable (const std::string &)

  Enable the state of a given state component.
- void setStateDisable (const std::string &)

  Disable the state of a given state component.
- stateType getState (const std::string &)

  Get the state of a given state component.
- void loadSpriteFromFile (std::string name, std::string filename)

  Load a sprite into the engine.
- void runMainLoop ()

  Run the main engine loop.

Static Public Member Functions

- static ArmyEngine * getInstance ()

  Singleton method used to retrieve the single instance.

4.4.1 Detailed Description

Army engine singleton facade. Frontend to the engine.

4.4.2 Member Function Documentation

4.4.2.1 static ArmyEngine* ArmyEngine::getInstance ( ) [inline], [static]

Singleton method used to retrieve the single instance.

Returns

Returns the army engine instance.

```
113                                     {
114     static ArmyEngine _instance;
115     return &_instance;
116   }
```

The documentation for this class was generated from the following files:

- ArmyEngine.h
- ArmyEngine.cpp

4.5 attribute_struct Struct Reference

structure used to represent the attribute

`#include <AbstractComponent.h>`

Public Attributes

- attribute_container **attr_container**
- attribute_type **attr_type**

4.5.1 Detailed Description

structure used to represent the attribute

The attribute is a structure which holds both the enumeration which describes the type of value stored, and the union container which holds the value.

Parameters

attr_container	Union holding the data that represents the attribute
attr_type	The type of the data being stored for the attribute

The documentation for this struct was generated from the following file:

- AbstractComponent.h

4.6 attributeContainer_union Union Reference

union container to store the attribute values

`#include <AbstractComponent.h>`

Public Attributes

- int i

  integer type
- float f

  float type
- std::string * s

  string type
- componentIntegerArrayType * si

  integer array type
- componentFloatArrayType * sf

  float array type

4.6.1 Detailed Description

union container to store the attribute values

The container holds the attribute value, and the type of value is described by the attribute_type

The documentation for this union was generated from the following file:

- AbstractComponent.h

4.7 CallbackFunctionWrapper Class Reference

Wrapper to store varying functions as one function type.

`#include <CallbackManager.h>`

Public Member Functions

- CallbackFunctionWrapper (functionTemplate func)

  Constructor to convert one-param to two-param.
- CallbackFunctionWrapper (functionBaseTemplate func)

  Constructor to convert two-param to two-param.
- int operator() (int ID, int eventIndex)

  operator() override

4.7.1 Detailed Description

Wrapper to store varying functions as one function type.

Wrap functionTemplates into functionEventTemplates Wrap functionBaseTemplates into functionEventTemplates

Used to wrap the other given function types into the functionEventTemplate type that is stored within the callback manager.

The wrapper converts zero-paramter and one-parameter functions into the two-paramter function used within the callback manager, and assigns the unused parameters as zero-values.

4.7.2 Member Function Documentation

4.7.2.1 int CallbackFunctionWrapper::operator() (int *ID,* int *eventIndex*)

operator() override

The function calls func_oneParam or func_zeroParam depending on the funcType of the assigned function.

Parameters

ID	is designated to the entity ID that performed the callback
eventIndex	is designated to the index of the event within the EventManager

Returns

A non-zero value if the callback was successful

```
49                                                                  {
50     if (this->funcType == ONE_PARAM) {
51         return this->func_oneParam(ID);
52     }
53     else if (this->funcType == ZERO_PARAM) {
54         return this->func_zeroParam();
55     }
56     return 1;
57 }
```

The documentation for this class was generated from the following files:

- CallbackManager.h
- CallbackManager.cpp

4.8 CallbackManager Class Reference

The callback manager is used to store and retrieve callbacks.

`#include <CallbackManager.h>`

Public Member Functions

- void addCallback (const std::string &name, functionBaseTemplate func)

  Add a callback to the callback manager with zero parameters.
- void addCallback (const std::string &name, functionTemplate func)

  Add a callback to the callback manager with one parameter.
- void addCallback (const std::string &name, functionEventTemplate func)

  Add a callback to the callback manager with one parameter.
- int triggerCallback (const std::string &name, int ID=0, int eventIndex=0)

  triggers the given callback, and returns whether it was successful
- void removeCallback (const std::string &name)

  deletes a callback by its key
- bool hasCallback (const std::string &name)

  Checks if the manager has the callback by the given keyname.

Static Public Member Functions

- static CallbackManager * getInstance ()

  < Singleton method to grab the entity manager instance.

4.8.1 Detailed Description

The callback manager is used to store and retrieve callbacks.

The callback manager contains lists of functions which are used as callbacks to components stored within the entities. It includes one parameter, which is the entities current ID the function must also return '0' in order to determine whether it was successful

4.8.2 Member Function Documentation

4.8.2.1 void CallbackManager::addCallback (const std::string & *name,* functionBaseTemplate *func*)

Add a callback to the callback manager with zero parameters.

add a callback to the map the function.

It is of the form int *f()*, and can be a lambda, std::bind, or class object, see the Unit Test CallbackTests

Parameters

name	is the keyname assigned to the callback
func	is the function you wish to execute for the callback

```
25                                                                          {
26     assert(!this->hasCallback(name) && "Callback already exists by that name");
27     auto funcWrapper = CallbackFunctionWrapper((
       functionBaseTemplate) func);
28     this->callbackMap[name] = funcWrapper;
29 }
```

4.8.2.2 void CallbackManager::addCallback (const std::string & *name,* functionTemplate *func*)

Add a callback to the callback manager with one parameter.

add a callback to the map the function.

It is of the form *int f(int)*, and can be a lambda, std::bind, or class object, see the Unit Test CallbackTests

Parameters

name	is the keyname assigned to the callback
func	is the function you wish to execute for the callback

```
17                                                                                {
18     assert(!this->hasCallback(name) && "Callback already exists by that name");
19
20     //wrap our function in our functionwrapper
21     auto funcWrapper = CallbackFunctionWrapper((
       functionTemplate) func);
22     this->callbackMap[name] = funcWrapper;
23 }
```

4.8.2.3 void CallbackManager::addCallback (const std::string & *name,* functionEventTemplate *func*)

Add a callback to the callback manager with one parameter.

add a callback to the map the function.

It is of the form *int f(int, int)*, and can be a lambda, std::bind, or class object, see the Unit Test CallbackTests

Parameters

name	is the keyname assigned to the callback
func	is the function you wish to execute for the callback

```
12                                                                                {
13     assert(!this->hasCallback(name) && "Callback already exists by that name");
14     this->callbackMap[name] = func;
15 }
```

4.8.2.4 static CallbackManager* CallbackManager::getInstance ( ) [inline], [static]

< Singleton method to grab the entity manager instance.

Returns

The callback manager instance.

```
89                                      {
90     static CallbackManager _instance;
91     return &_instance;
92   }
```

4.8.2.5 bool CallbackManager::hasCallback (const std::string & *name*)

Checks if the manager has the callback by the given keyname.

returns true, if the provided parameter matches a callback key stored within the callbackMap

Parameters

name	is the keyname of the callback you wish to check.

Returns

The method returns true if the callback exists, otherwise it returns false.

```
5                                                   {
6     if (this->callbackMap.find(name) == this->callbackMap.end()) {
7         return false;
```

```
8       }
9       return true;
10  }
```

4.8.2.6 void CallbackManager::removeCallback (const std::string & *name*)

deletes a callback by its key

Parameters

name	is the keyname of the callback to remove.

```
36                                                      {
37      assert(this->hasCallback(name) && "Callback does not exist by that name");
38      this->callbackMap.erase(name);
39  }
```

4.8.2.7 int CallbackManager::triggerCallback (const std::string & *name*, int *ID* = 0, int *eventIndex* = 0)

triggers the given callback, and returns whether it was successful

Parameters

name	is the keyname of the callback to call
ID	is the first parameter, which is usually designated to the ID of an entity.
eventIndex	is the second parameter, which is usually designated to the index of the event within the EventManager.

```
31                                                                          {
32      assert(this->hasCallback(name) && "Callback does not exist by that name");
33      return this->callbackMap[name](ID, eventIndex);
34  }
```

The documentation for this class was generated from the following files:

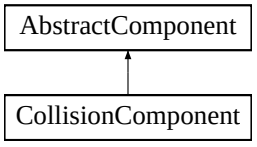
- CallbackManager.h
- CallbackManager.cpp

4.9 CollisionComponent Class Reference

The collision component is used to assign a collision bound to an entity.

`#include <CollisionComponent.h>`

Inheritance diagram for CollisionComponent:



Public Member Functions

- CollisionComponent (std::string name)

  Constructor for the collision component.

- int update ()

  Dynamically bound function that updates the component.

4.9.1 Detailed Description

The collision component is used to assign a collision bound to an entity.

The collision component includes attributes to determine the positioning of a figmentary shape on the screen that represents the bounds of that given entity.

All attributes that contribute to the offset, origin and position of a shape component also have the same attributes associated with the collision component.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 CollisionComponent::CollisionComponent (std::string *name*)

Constructor for the collision component.

The constructor for the collision component overrides the family to the fixed "Collision" family name. It is recommended that no other components created override this name.

Parameters

name	is the unique name assigned to the given collision component.

```
5                                                   :
6     AbstractComponent(name, "Collision") {
7     //Determines whether the given shape should be shown
8         //0 - disable
9         //1 - enable
10     setAttribute_int(ATTRIBUTE_ENABLE, 1);
11
12     //the collision tag, which is an identifier for determining the collision
13      //provides the manager with unique collisions to provide callbacks for
14     setAttribute_string(ATTRIBUTE_COLLISION_TAG,
      COLLISION_DEFAULT_TAG);
15
16     //the collision bound shape type
17     setAttribute_string(ATTRIBUTE_COLLISION_BOUND_TYPE,
      COLLISION_BOUND_RECTANGLE);
18
19     //the collision bound attributes
20     //  rectangle depends on width and height
21     //  circle depends on radius
22
23     //width of the rectangle bound (x)
24     setAttribute_float(ATTRIBUTE_WIDTH, 0.0);
25     //height of the rectangle bound (y)
26     setAttribute_float(ATTRIBUTE_HEIGHT, 0.0);
27     //radius of the circle bound (r)
28     setAttribute_float(ATTRIBUTE_RADIUS, 0.0);
29     //Used by polygons to represent the points making up the bounds
30     setAttribute_floatArray(ATTRIBUTE_POLYGON_POINTS,
      componentFloatArrayType());
31
32     //Related Sprite attributes
33     //This provides the offset of the collision bound from its origin within the X and Y direction
34     this->setAttribute_float(ATTRIBUTE_OFFSET_X, 0.0);
35     this->setAttribute_float(ATTRIBUTE_OFFSET_Y, 0.0);
36
37     //Is the origin of the collision bound with respect to the
38       //position component
39     this->setAttribute_float(ATTRIBUTE_ORIGIN_X, 0.0);
40     this->setAttribute_float(ATTRIBUTE_ORIGIN_Y, 0.0);
41
}
```

The documentation for this class was generated from the following files:

- CollisionComponent.h
- CollisionComponent.cpp

4.10 CollisionManager Class Reference

Collision manager used to perform callbacks on collisions.

`#include <CollisionManager.h>`

Public Member Functions

- void addCallback (const collisionTagTuple &tagTuple, functionCollisionTemplate func)

  used to add a callback to the callback table
- int triggerCallback (const collisionTagTuple &tagTuple, const collisionParamTuple &paramTuple)

  called by the collision system to trigger a desired callback
- void removeCallback (const collisionTagTuple &tagTuple)

  deletes the callback described by the provided tuple
- bool hasCallback (const collisionTagTuple &tagTuple)

  checks if the callback table has the provided tag
- void registerCollision (const registeredCollisionTuple &theTuple)

  registers the collision within the list
- void unregisterCollision (const registeredCollisionTuple &theTuple)

  unregisters and removes from the list
- bool hasRegisteredCollision (const registeredCollisionTuple &theTuple)

  checks to see if the current collision is registered

Static Public Member Functions

- static CollisionManager * getInstance ()

  Singleton method, returns an instance of the manager.

4.10.1 Detailed Description

Collision manager used to perform callbacks on collisions.

Collision manager is used to manage what happens when two entities collide with eachother. Callbacks are added to the collision manager to handle situations where given collision types collide. Depending on the type of collision, the resulting collision may be handled differently.

4.10.2 Member Function Documentation

4.10.2.1 void CollisionManager::addCallback (const collisionTagTuple & *tagTuple*, functionCollisionTemplate *func*)

used to add a callback to the callback table

Parameters

tagTuple	is two tags representing the type of collision performed.
func	is the function to assign as a callback when the collision is performed.

```
7                                                                                                {
8     assert(!this->hasCallback(tagTuple) && "Already has the given relationship");
9     this->callbackTable[tagTuple] = func;
10 }
```

4.10.2.2 static CollisionManager* CollisionManager::getInstance ( ) [inline], [static]

Singleton method, returns an instance of the manager.

Returns

The instance of the CollisionManager

```
132                                         {
133     static CollisionManager _instance;
134     return &_instance;
135   }
```

4.10.2.3 bool CollisionManager::hasCallback (const collisionTagTuple & *tagTuple*)

checks if the callback table has the provided tag

Parameters

tagTuple	is two tags representing the type of collision performed.

```
12                                                                  {
13     if (this->callbackTable.find(tagTuple) == this->callbackTable.end()) {
14         return false;
15     }
16     return true;
17 }
```

4.10.2.4 bool CollisionManager::hasRegisteredCollision (const registeredCollisionTuple & *theTuple*)

checks to see if the current collision is registered

Parameters

theTuple	is a unique description for a collision performed between two entities

```
39                                                                                  {
40     auto iter = std::find(this->registeredCollisionList.begin(),
41         this->registeredCollisionList.end(),
42         theTuple);
43     if (iter == this->registeredCollisionList.end()) {
44         return false;
45     }
46     return true;
47 }
```

4.10.2.5 void CollisionManager::registerCollision (const registeredCollisionTuple & *theTuple*)

registers the collision within the list

Parameters

theTuple	is a unique description for a collision performed between two entities

```
29                                                                      {
30     assert(!this->hasRegisteredCollision(theTuple) && "theTuple already exists");
31     this->registeredCollisionList.push_back(theTuple);
32 }
```

4.10.2.6 int CollisionManager::triggerCallback (const collisionTagTuple & *tagTuple*, const collisionParamTuple & *paramTuple*)

called by the collision system to trigger a desired callback

Parameters

tagTuple	is two tags representing the type of collision performed.
paramTuple	is the parameters to bind to the given callback described by the tagTuple.

```
19
        {
20      assert(this->hasCallback(tagTuple) && "Can't trigger callback, callback doesn't exist");
21      return this->callbackTable[tagTuple](paramTuple);
22  }
```

4.10.2.7 void CollisionManager::unregisterCollision (const registeredCollisionTuple & *theTuple*)

unregisters and removes from the list

Parameters

theTuple	is a unique description for a collision performed between two entities

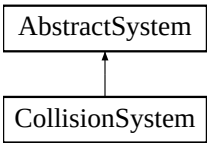
```
34                                                                  {
35      assert(this->hasRegisteredCollision(theTuple) && "theTuple doesn't exist");
36      this->registeredCollisionList.remove(theTuple);
37  }
```

The documentation for this class was generated from the following files:

- CollisionManager.h
- CollisionManager.cpp

4.11 CollisionSystem Class Reference

Inheritance diagram for CollisionSystem:



Public Member Functions

- int process ()

  Processes the entities with collision components.

Additional Inherited Members

4.11.1 Member Function Documentation

4.11.1.1 int CollisionSystem::process ( ) [virtual]

Processes the entities with collision components.

This method is run on each frame.

It checks each of the entities for a collision component and a position component. And performs checks between entities that could potentialy perform a collision callback. If they are checked and it turns out they are colliding, the callback is triggered with information pertaining to the two entities affected.

Implements AbstractSystem.

```
564                             {
565     //need to grab all of the entities that include a collision components
566     std::vector<std::shared_ptr<entityType>> collidableEntities;
567     for (auto entity : this->entityManager->getAllEntities()) {
568         if ((!entity->hasComponentFamily("Collision") ||
569             !entity->hasComponentFamily("Position"))) {
570             continue;
571         }
572         else {
573             collidableEntities.push_back(entity);
574         }
575     } //END for (auto entity : entityManager->getAllEntities()) {
576 
577     //testing collision components to eachother.
578     //the robustness of the test depends on what types of bounded objects will end up colliding.
579     //need to form tests between
580       //Bounded Rectangle
581       //Bounded Circle
582       //Bounded Triangle
583       //Bounded Polygon (not implemented)
584     for (auto entityFirst : collidableEntities) {
585         auto positionComponentFirst = entityFirst->getComponentsByFamily("Position").front();
586         for (auto collisionComponentFirst : entityFirst->getComponentsByFamily("Collision")) {
587         auto collisionBoundTypeFirst = collisionComponentFirst->getAttribute_string(
    ATTRIBUTE_COLLISION_BOUND_TYPE);
588         auto collisionTagFirst = collisionComponentFirst->getAttribute_string(
    ATTRIBUTE_COLLISION_TAG);
589 
590         //continue if the given collision component isn't enabled
591         if (!(collisionComponentFirst->getAttribute_int(ATTRIBUTE_ENABLE))) {
592             continue;
593         }
594 
595         for (auto entitySecond : collidableEntities) {
596             //don't compare to itself
597             if (entityFirst->getID() == entitySecond->getID()) {
598                 continue;
599             }
600 
601             //assuming that our entities only have one
602             //collision component and one position component
603             //  need to consider doing collision hierarchies later
604             auto positionComponentSecond = entitySecond->getComponentsByFamily("Position").front();
605             for (auto collisionComponentSecond : entitySecond->getComponentsByFamily("Collision")) {
606 
607             //continue if it isn't enabled
608             if (!(collisionComponentSecond->getAttribute_int(ATTRIBUTE_ENABLE))) {
609                 continue;
610             }
611 
612             auto collisionBoundTypeSecond = collisionComponentSecond->getAttribute_string(
    ATTRIBUTE_COLLISION_BOUND_TYPE);
613             auto collisionTagSecond = collisionComponentSecond->getAttribute_string(
    ATTRIBUTE_COLLISION_TAG);
614 
615             //make the tagTuple
616             auto tagTuple = collisionTagTuple(collisionTagFirst, collisionTagSecond);
617 
618             //no point in checking for collisions if their is no registered callback
619             if (!collisionManager->hasCallback(tagTuple)) {
620                 continue;
621             }
622 
623             //check to see if they have collided
624             //perform collision tests for each unique case
625             bool bHasCollided = false;
626 
627             //collision between two rectangles
628             if (collisionBoundTypeFirst == COLLISION_BOUND_RECTANGLE &&
629                 collisionBoundTypeSecond == COLLISION_BOUND_RECTANGLE) {
630                 bHasCollided = this->collision_rect_rect(
631                     positionComponentFirst,
632                     collisionComponentFirst,
633                     positionComponentSecond,
634                     collisionComponentSecond
635                     );
636             }
```

```
637             //collision between two circles
638             else if (collisionBoundTypeFirst == COLLISION_BOUND_CIRCLE &&
639                 collisionBoundTypeSecond == COLLISION_BOUND_CIRCLE) {
640                 bHasCollided = this->collision_circle_circle(
641                     positionComponentFirst,
642                     collisionComponentFirst,
643                     positionComponentSecond,
644                     collisionComponentSecond
645                     );
646             }
647 
648             //collision between a rectangle and a circle
649             else if (collisionBoundTypeFirst == COLLISION_BOUND_RECTANGLE &&
650                 collisionBoundTypeSecond == COLLISION_BOUND_CIRCLE) {
651                 bHasCollided = this->collision_rect_circle(
652                     positionComponentFirst,
653                     collisionComponentFirst,
654                     positionComponentSecond,
655                     collisionComponentSecond
656                     );
657             }
658             else if (collisionBoundTypeFirst == COLLISION_BOUND_CIRCLE &&
659                 collisionBoundTypeSecond == COLLISION_BOUND_RECTANGLE) {
660                 //swap the parameters
661                 bHasCollided = this->collision_rect_circle(
662                     positionComponentSecond,
663                     collisionComponentSecond,
664                     positionComponentFirst,
665                     collisionComponentFirst
666                     );
667             }
668             else if (collisionBoundTypeFirst == COLLISION_BOUND_RECTANGLE &&
669                 collisionBoundTypeSecond == COLLISION_BOUND_POLYGON) {
670                 bHasCollided = this->collision_rect_polygon(
671                     positionComponentFirst,
672                     collisionComponentFirst,
673                     positionComponentSecond,
674                     collisionComponentSecond
675                     );
676             }
677             else if (collisionBoundTypeFirst == COLLISION_BOUND_POLYGON &&
678                 collisionBoundTypeSecond == COLLISION_BOUND_RECTANGLE) {
679                     //swap the parameters
680                 bHasCollided = this->collision_rect_polygon(
681                     positionComponentSecond,
682                     collisionComponentSecond,
683                     positionComponentFirst,
684                     collisionComponentFirst
685                     );
686             }
687             else if (collisionBoundTypeFirst == COLLISION_BOUND_POLYGON &&
688                 collisionBoundTypeSecond == COLLISION_BOUND_POLYGON) {
689                 bHasCollided = this->collision_polygon_polygon(
690                     positionComponentFirst,
691                     collisionComponentFirst,
692                     positionComponentSecond,
693                     collisionComponentSecond
694                     );
695             }
696             else if (collisionBoundTypeFirst == COLLISION_BOUND_CIRCLE &&
697                 collisionBoundTypeSecond == COLLISION_BOUND_POLYGON) {
698                 bHasCollided = this->collision_circle_polygon(
699                     positionComponentFirst,
700                     collisionComponentFirst,
701                     positionComponentSecond,
702                     collisionComponentSecond
703                     );
704             }
705             else if (collisionBoundTypeFirst == COLLISION_BOUND_POLYGON &&
706                 collisionBoundTypeSecond == COLLISION_BOUND_CIRCLE) {
707                 bHasCollided = this->collision_circle_polygon(
708                     positionComponentSecond,
709                     collisionComponentSecond,
710                     positionComponentFirst,
711                     collisionComponentFirst
712                     );
713             }
714 
715 
716             //make the registered collision tuple
717             auto regTuple = registeredCollisionTuple(
718                 entityFirst->getID(), collisionComponentFirst->getName(),
719                 entitySecond->getID(), collisionComponentSecond->getName()
720                 );
721 
722             //if they have collided, trigger a callback if a callback exists by the given tag pair
723             //if it's already been registered, then we can simply skip over it
```

```
724             if (bHasCollided) {
725                 //make the tag pair
726                 if (collisionManager->hasCallback(tagTuple)) {
727                     //we only need to call the collision callback once, so if it's
728                     // already registered, there is no point in calling it again
729                     if (this->collisionManager->hasRegisteredCollision(regTuple)) {
730                         continue;
731                     }
732                     else {
733                         this->collisionManager->registerCollision(regTuple);
734                         //create our param tuple
735                         collisionParamTuple paramTuple = std::tuple_cat(regTuple,
    std::tuple<bool>(true));
736                         collisionManager->triggerCallback(tagTuple, paramTuple);
737                     }
738 
739                 }
740             }
741             else { //if (!bHasCollided)
742                 if (collisionManager->hasRegisteredCollision(regTuple)) {
743                     //need to unregister and perform the callback with a false boolean
744                     this->collisionManager->unregisterCollision(regTuple);
745                     collisionParamTuple paramTuple = std::tuple_cat(regTuple,
    std::tuple<bool>(false));
746                     collisionManager->triggerCallback(tagTuple, paramTuple);
747                 }
748             }
749 
750         } //END for (auto collisionComponentSecond : entitySecond->getComponentsByFamily("Collision")) {
751         } //END for (auto entity2 : collidableEntities) {
752     } //END for (auto collisionComponentFirst : entityFirst->getComponentsByFamily("Collision")) {
753     } //END for (auto entity1 : collidableEntities) {
754     return 0;
755 }
```

The documentation for this class was generated from the following files:

- CollisionSystem.h
- CollisionSystem.cpp

4.12 ComponentFactory Class Reference

Public Member Functions

- std::shared_ptr< componentType > createCollisionComponent (std::string name)

  Used to create CollisionComponent.
- std::shared_ptr< componentType > createHealthComponent (std::string name)

  Used to create HealthComponent.
- std::shared_ptr< componentType > createInputComponent (std::string name)

  Used to create InputComponent.
- std::shared_ptr< componentType > createPositionComponent (std::string name)

  Used to create PositionComponent.
- std::shared_ptr< componentType > createShapeComponent (std::string name)

  Used to create ShapeComponent.
- std::shared_ptr< componentType > createSpriteComponent (std::string name)

  Used to create SpriteComponent.
- std::shared_ptr< componentType > createStateComponent (std::string name)

  Used to create StateComponent.
- std::shared_ptr< componentType > createTextComponent (std::string name)

  Used to create TextComponent.

4.12.1 Member Function Documentation

4.12.1.1 std::shared_ptr< componentType > ComponentFactory::createCollisionComponent (std::string *name*)

Used to create CollisionComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
7                                                                                {
8     return std::shared_ptr<componentType>(new CollisionComponent(name));
9 }
```

4.12.1.2 std::shared_ptr< componentType > ComponentFactory::createHealthComponent (std::string *name*)

Used to create HealthComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
11                                                                               {
12     return std::shared_ptr<componentType>(new HealthComponent(name));
13 }
```

4.12.1.3 std::shared_ptr< componentType > ComponentFactory::createInputComponent (std::string *name*)

Used to create InputComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
15                                                                               {
16     return std::shared_ptr<componentType>(new InputComponent(name));
17 }
```

4.12.1.4 std::shared_ptr< componentType > ComponentFactory::createPositionComponent (std::string *name*)

Used to create PositionComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
19                                                                          {
20     return std::shared_ptr<componentType>(new PositionComponent(name));
21 }
```

4.12.1.5 std::shared_ptr< componentType > ComponentFactory::createShapeComponent (std::string *name*)

Used to create ShapeComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
23                                                                          {
24     return std::shared_ptr<componentType>(new ShapeComponent(name));
25 }
```

4.12.1.6 std::shared_ptr< componentType > ComponentFactory::createSpriteComponent (std::string *name*)

Used to create SpriteComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
27                                                                          {
28     return std::shared_ptr<componentType>(new SpriteComponent(name));
29 }
```

4.12.1.7 std::shared_ptr< componentType > ComponentFactory::createStateComponent (std::string *name*)

Used to create StateComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
31                                                                          {
32     return std::shared_ptr<componentType>(new StateComponent(name));
33 }
```

4.12.1.8 std::shared_ptr< componentType > ComponentFactory::createTextComponent (std::string *name*)

Used to create TextComponent.

Parameters

name	is the unique name to be assigned to the component

Returns

Returns the newly created component.

```
35                                                                                  {
36     return std::shared_ptr<componentType>(new TextComponent(name));
37 }
```

The documentation for this class was generated from the following files:

- ComponentFactory.h
- ComponentFactory.cpp

4.13 EntityFactory Class Reference

Factory for creating entities.

`#include <EntityFactory.h>`

Public Member Functions

- std::shared_ptr< entityType > createMainEntity (std::string name)

  Returns a newly created Main entity.

4.13.1 Detailed Description

Factory for creating entities.

4.13.2 Member Function Documentation

4.13.2.1 std::shared_ptr< entityType > EntityFactory::createMainEntity (std::string *name*)

Returns a newly created Main entity.

Parameters

name	is the unique name assigned to the entity.

Returns

Returns the newly created entity.

```
8                                                                   {
9     auto entity = std::shared_ptr<entityType>(new MainEntity(name, EntityFactory::idNum));
10     EntityFactory::idNum += 1;
11     return entity;
12 }
```

The documentation for this class was generated from the following files:

- EntityFactory.h
- EntityFactory.cpp

4.14 EntityManager Class Reference

The entity manager used to store and manager entities.

`#include <EntityManager.h>`

Public Member Functions

- void addEntity (std::shared_ptr< entityType > entity)

  Add entity to the entity manager.
- void removeEntity (std::shared_ptr< entityType > entity)

  Remove entity from the entity manager.
- bool hasEntityById (const int ID)

  Checks if an entity with the given ID exists within the manager.
- std::shared_ptr< entityType > getEntityById (const int ID)

  Gets the first entity with the given unique ID.
- entityVectorType getAllEntities ()

  Gets all of the entities within the manager.
- entityListType getEntitiesByName (const std::string &entityName)

  Gets all of the entities by the given name.
- entityListType getEntitiesByFamily (const std::string &entityFamily)

  Gets all of the entities with the given family.
- void sortEntityList ()

  operation sorts the entityList based on a sorting criteria

Static Public Member Functions

- static EntityManager * getInstance ()

  Singleton method to grab the entity manager instance.

4.14.1 Detailed Description

The entity manager used to store and manager entities.

The entity manager is a singleton that has methods to add / store, remove, retrieve one or many entities. Any entities that you want to be processed within the engine should be added to the entity manager.

4.14.2 Member Function Documentation

4.14.2.1 void EntityManager::addEntity (std::shared_ptr< entityType > *entity*)

Add entity to the entity manager.

Used to add an entity to the entity manager. The entity needs to be in the form of a shared_ptr.

Parameters

entity	is the shared_ptr of the entity you wish to add.

```
12                                                          {
13      this->entityList.push_back(entity);
14  }
```

4.14.2.2 entityVectorType EntityManager::getAllEntities ( )

Gets all of the entities within the manager.

Returns

The list of all the entities.

```
43                                                  {
44      return this->entityList;
45  }
```

4.14.2.3 entityListType EntityManager::getEntitiesByFamily (const std::string & *entityFamily*)

Gets all of the entities with the given family.

This list returned will contain all entities that belong to the same family.

Parameters

entityFamily	is the family you wish to retrieve

Returns

The method returns a list of entities that are a part of the given family.

```
57                                                                          {
58      std::list<std::shared_ptr<entityType>> entityList;
59      for ( auto entity : this->entityList) {
60          if (entityFamily == entity->getFamily()) {
61              entityList.push_back(entity);
62          }
63      }
64      return entityList;
65  }
```

4.14.2.4 entityListType EntityManager::getEntitiesByName (const std::string & *entityName*)

Gets all of the entities by the given name.

This list returned will contain all of the entities with the given name. This suggests that entities do not have a unique name, which would be correct. TODO: fix this.

Parameters

entityName	is the name of the entities you wish to retrieve.

Returns

The method returns a list of entities with the given name

```
47                                                                      {
48      std::list<std::shared_ptr<entityType>> entityList;
49      for ( auto entity : this->entityList) {
50          if (entityName == entity->getName()) {
51              entityList.push_back(entity);
52          }
53      }
```

```
54     return entityList;
55 }
```

4.14.2.5 std::shared_ptr< entityType > EntityManager::getEntityById (const int *ID*)

Gets the first entity with the given unique ID.

Parameters

ID	is the unique identifier for the entity

Returns

The entity, or a shared_ptr(nullptr) if the entity does not exist.

```
33                                                                  {
34     assert(this->hasEntityById(ID) && "No entity with the given ID");
35     for (auto entity : this->entityList) {
36         if (ID == entity->getID()) {
37             return entity;
38         }
39     }
40     return std::shared_ptr<entityType> (nullptr);
41 }
```

4.14.2.6 static EntityManager* EntityManager::getInstance ( ) [inline], [static]

Singleton method to grab the entity manager instance.

Returns

The entity manager instance.

```
71                                      {
72     static EntityManager _instance;
73     return &_instance;
74   }
```

4.14.2.7 bool EntityManager::hasEntityById (const int *ID*)

Checks if an entity with the given ID exists within the manager.

Parameters

ID	is the unique identifier for the entity

Returns

Returns true if the entity with the given ID exists, otherwise it returns false.

```
24                                              {
25     for (auto entity : this->entityList) {
26         if (ID == entity->getID()) {
27             return true;
28         }
29     }
30     return false;
31 }
```

4.14.2.8 void EntityManager::removeEntity (std::shared_ptr< entityType > *entity*)

Remove entity from the entity manager.

Used to remove entity from the entity manager. This does not necessarily deallocate the entity if references are still made between with the shared_ptr.

Parameters

entity	is the shared_ptr of the entity.

```
16                                                                  {
17     assert(this->hasEntityById(entity->getID()) && "No entity exists with the given ID");
18     auto iter = std::find(this->entityList.begin(), this->entityList.end(), entity);
19     if (iter != this->entityList.end()) {
20         this->entityList.erase(iter);
21     }
22 }
```

4.14.2.9 void EntityManager::sortEntityList ( )

operation sorts the entityList based on a sorting criteria

The entityList stored within the entity manager is currently being sorted baed on the lowest Z-Buffer value for components stored within each entity. Given the large number of entities that would likely be present, this would present a very unwelcome performance hit if it is called each frame.

```
67                                  {
68     //call the handle on each of our entities
69     for (auto entity : this->entityList) {
70         entity->handle();
71     }
72
73     std::sort(entityList.begin(), entityList.end(),
74         [] (std::shared_ptr<entityType> first, std::shared_ptr<entityType> second) {
75             auto componentVector = first->getAllComponents();
76             //grab first component with a z-buffer,
77             // assuming components were sorted by entity.handle() function
78             // for this particular situation
79             auto compIter = std::find_if(componentVector.begin(), componentVector.end(),
80                 [] (std::shared_ptr<componentType> elem) {
81                     if (elem->hasAttribute(ATTRIBUTE_ZBUFFER)) {
82                         return true;
83                     }
84                     return false;
85             });
86             //if our first entity doesn't have a component
87             // with a Z-buffer, we return false
88             if (compIter == componentVector.end()) {
89                 return false;
90             }
91             //grab our Z-Buffer value from the component stored within
92             // the iterator
93             auto componentFirst = *compIter;
94             float firstZ = componentFirst->getAttribute_float(ATTRIBUTE_ZBUFFER);
95
96             componentVector = second->getAllComponents();
97             compIter = std::find_if(componentVector.begin(), componentVector.end(),
98                 [] (std::shared_ptr<componentType> elem) {
99                     if (elem->hasAttribute(ATTRIBUTE_ZBUFFER)) {
100                         return true;
101                     }
102                     return false;
103             });
104
105             //if our second entity doesn't have a component
106             // with a Z-buffer, we return true
107             if (compIter == componentVector.end()) {
108                 return true;
109             }
110             //grab our Z-Buffer value from the component stored within
111             // the iterator
112             auto componentSecond = *compIter;
113             float secondZ = componentSecond->getAttribute_float(
    ATTRIBUTE_ZBUFFER);
114
115             return (firstZ < secondZ);
```

```
116     });
117 }
```

The documentation for this class was generated from the following files:

- EntityManager.h
- EntityManager.cpp

4.15 EventManager Class Reference

Used to Poll for events and store them for later retrieval each frame.

`#include <EventManager.h>`

Public Member Functions

- void setWindow (sf::RenderWindow *window)

  Sets the current window to retrieve events from.
- void pollEvents ()

  polls the assigned window for events and stores them.
- eventListType & getEvents ()

  Gets the list of events from the event manager.

Static Public Member Functions

- static EventManager * getInstance ()

  Singletone method, returns the single instance.

4.15.1 Detailed Description

Used to Poll for events and store them for later retrieval each frame.

The Event manager polls for events and stores them. The current implementation stores events only from the SFML event polling system, which will be changed in the future.

4.15.2 Member Function Documentation

4.15.2.1 eventListType & EventManager::getEvents ( )

Gets the list of events from the event manager.

Returns

The list of events.

```
28                                  {
29     assert(this->window != nullptr && "sf::RenderWindow instance must be assigned to EventManager");
30     return this->eventList;
31 }
```

4.15.2.2 static EventManager* EventManager::getInstance ( ) [inline], [static]

Singletone method, returns the single instance.

Returns

Returns the single instance of the event manager

```
83                                      {
84     static EventManager _instance;
85     return &_instance;
86   }
```

4.15.2.3 void EventManager::pollEvents ( )

polls the assigned window for events and stores them.

The event manager clears the list, polls the events that have occured and stores them within the event manager.

```
14                                {
15     assert(this->window != nullptr && "sf::RenderWindow instance must be assigned to EventManager");
16
17     //first, we clear out whatever events were in the list before
18     this->eventList.clear();
19
20     eventType event;
21     while(this->window->pollEvent(event)) {
22         auto allocEvent = std::shared_ptr<eventType> (new eventType);
23         *allocEvent = event;
24         this->eventList.push_back(allocEvent);
25     }
26 }
```

4.15.2.4 void EventManager::setWindow (sf::RenderWindow * *window*)

Sets the current window to retrieve events from.

Parameters

window	is the window we wish to retreive events from.

```
10                                                      {
11     this->window = window;
12 }
```

The documentation for this class was generated from the following files:

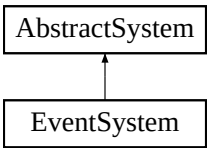
- EventManager.h
- EventManager.cpp

4.16 EventSystem Class Reference

Used to handle global events that affect the entire application.

`#include <EventSystem.h>`

Inheritance diagram for EventSystem:



Public Member Functions

- void registerClosed_Callback (functionEventTemplate)

  Assign callback for closing the window.
- void registerResized_Callback (functionEventTemplate)

  Assign callback for resizing the window.
- void registerLostFocus_Callback (functionEventTemplate)

  Assign callback for losing focus to the window.
- void registerGainedFocus_Callback (functionEventTemplate)

  Assign callback for gaining focus to the window.
- void registerTextEntered_Callback (functionEventTemplate)

  Assign callback for entering text into the window.
- void registerMouseEntered_Callback (functionEventTemplate)

  Assign callback for the mouse entering the window.
- void registerMouseLeft_Callback (functionEventTemplate)

  Assign callback for the mouse leaving the window.
- int process ()

  Used to process the events and perform the appropriate callbacks.

Additional Inherited Members

4.16.1 Detailed Description

Used to handle global events that affect the entire application.

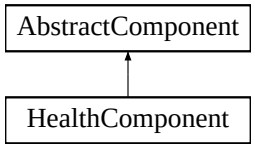
Event system is used to add event callbacks to handle certain events that are performed which might have a global impact on the application.

The documentation for this class was generated from the following files:

- EventSystem.h
- EventSystem.cpp

4.17 HealthComponent Class Reference

Inheritance diagram for HealthComponent:



Public Member Functions

- **HealthComponent** (std::string)
- int update ()

  Updates the component based on the current set of attributes.

4.17.1 Member Function Documentation

4.17.1.1 int HealthComponent::update ( ) [virtual]

Updates the component based on the current set of attributes.

Used to update the component. This should be performed after modifications have been made on the component's attributes or members.

Returns

Returns a non-zero value if it is successful.

Implements AbstractComponent.

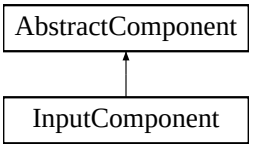
```
8                               {
9     return 0;
10 }
```

The documentation for this class was generated from the following files:

- HealthComponent.h
- HealthComponent.cpp

4.18 InputComponent Class Reference

Inheritance diagram for InputComponent:



Public Member Functions

- **InputComponent** (std::string)
- int update ()

  Updates the component based on the current set of attributes.

4.18.1 Member Function Documentation

4.18.1.1 int InputComponent::update ( ) [virtual]

Updates the component based on the current set of attributes.

Used to update the component. This should be performed after modifications have been made on the component's attributes or members.

Returns

Returns a non-zero value if it is successful.

Implements AbstractComponent.

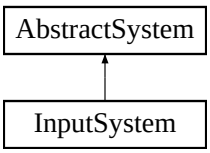
```
26                              {
27     return 0;
28 }
```

The documentation for this class was generated from the following files:

- InputComponent.h
- InputComponent.cpp

4.19 InputSystem Class Reference

Inheritance diagram for InputSystem:



Public Member Functions

- int process ()

  Perform actions on entities.

Additional Inherited Members

4.19.1 Member Function Documentation

4.19.1.1 int InputSystem::process ( ) [virtual]

Perform actions on entities.

The process method is a pure virtual function that processes all of th entities stored within the entity manager. Some systems do not use the entity manager for processing, but may process other things that require processing per frame.

What is processed is fully dependant on the implementation.

Returns

A non-zero value if the processing was successful.

Implements AbstractSystem.

```
23                          {
24
25     //we need to sift through the entities
26     for(auto entity : this->entityManager->getAllEntities()) {
27         //determine if the entity has what we desire
28         if(!entity->hasComponentFamily("Input")) {
29             continue;
30         }
```

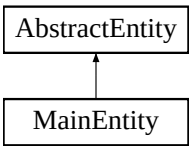
```
31
32          //grab all of the events
33          auto eventList = this->eventManager->getEvents();
34
35          //grab all of the input components
36          for (auto inputComponent : entity->getComponentsByFamily("Input")) {
37              //check to see if the input is enabled
38              int bEnabled = inputComponent->getAttribute_int(ATTRIBUTE_ENABLE);
39              if (bEnabled == 0) {
40                  continue;
41              }
42
43              //get the input type
44              std::string inputType = inputComponent->getAttribute_string("InputType");
45
46              for (auto event : eventList) {
47                  bool bIsEvent = false;
48                  if (inputType == INPUT_MOUSE_MOVE) {
49                      bIsEvent = this->isEvent(*event, inputType);
50                  }
51                  else if (inputType == INPUT_MOUSE_PRESSED) {
52                      bIsEvent = this->isEvent(*event, inputType);
53                  }
54                  else if (inputType == INPUT_MOUSE_RELEASED) {
55                      bIsEvent = this->isEvent(*event, inputType);
56                  }
57                  else if (inputType == INPUT_MOUSE_SCROLL) {
58                      bIsEvent = this->isEvent(*event, inputType);
59                  }
60                  else if (inputType == INPUT_KEYBOARD_PRESSED) {
61                      bIsEvent = this->isEvent(*event, inputType);
62                  }
63                  else if (inputType == INPUT_KEYBOARD_RELEASED) {
64                      bIsEvent = this->isEvent(*event, inputType);
65                  }
66                  else {
67                      continue;
68                  }
69
70                  if (bIsEvent) {
71                      //upon finding a matching event to the input, grab the callback,
72                        //and trigger it with the given component
73                      //get ID
74                      int entityID = entity->getID();
75
76                      //get eventIndex
77                      auto iter = std::find(eventList.begin(), eventList.end(), event);
78                      assert(!(iter == eventList.end()) && "Event doesn't exist");
79                      int eventIndex = iter - eventList.begin();
80
81                      //get the callback from the inputComponent
82                      std::string callbackString = inputComponent->getAttribute_string(
    ATTRIBUTE_CALLBACK);
83
84                      //trigger the callback
85                      assert(callbackManager->hasCallback(callbackString) && "inputComponent
     callback doesn't exist");
86                      callbackManager->triggerCallback(callbackString, entityID, eventIndex);
87                  } //if (bIsEvent) { ...
88              }//END for (auto event : ...
89          } //END for (auto inputComponent : ...
90      } //END for(auto entity : ...
91      return 0;
92  }
```

The documentation for this class was generated from the following files:

- InputSystem.h
- InputSystem.cpp

4.20 MainEntity Class Reference

Inheritance diagram for MainEntity:



Public Member Functions

- **MainEntity** (std::string, int)
- int handle ()

  virtual function to perform pre-initialization and handling

4.20.1 Member Function Documentation

4.20.1.1 int MainEntity::handle ( ) [virtual]

virtual function to perform pre-initialization and handling

Currently it is being used to sort the shape, text, and sprite components to provide Z-buffer ordering. It is advised that this method be used if any components are added during execution to prevent erroneous behaviour.

Returns

A non-zero value when it is successful.

Implements AbstractEntity.

```
14                      {
15     auto componentVector = this->getAllComponents();
16     std::sort(componentVector.begin(), componentVector.end(),
17         [] (std::shared_ptr<componentType> first, std::shared_ptr<componentType> second) {
18             if (!first->hasAttribute(ATTRIBUTE_ZBUFFER)) {
19                 return false;
20             }
21             if (!second->hasAttribute(ATTRIBUTE_ZBUFFER)) {
22                 return true;
23             }
24
25             float zFirst = first->getAttribute_float(ATTRIBUTE_ZBUFFER);
26             float zSecond = second->getAttribute_float(ATTRIBUTE_ZBUFFER);
27             return (zFirst < zSecond);
28     });
29     return 0;
30 }
```

The documentation for this class was generated from the following files:

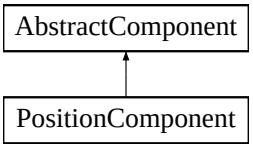
- MainEntity.h
- MainEntity.cpp

4.21 PositionComponent Class Reference

Used to describe the position of an entity.

`#include <PositionComponent.h>`

Inheritance diagram for PositionComponent:



Public Member Functions

- PositionComponent (std::string name)

  constructor for position component
- int update ()

  updates the position component. Not implemented.

4.21.1 Detailed Description

Used to describe the position of an entity.

The position of the entity is important for the SpriteComponent, TextComponent, ShapeComponent and Collision Component.

4.21.2 Constructor & Destructor Documentation

4.21.2.1 PositionComponent::PositionComponent (std::string *name*)

constructor for position component

The position component overrides the family value as "Position" and should not be assigned to any further components.

Parameters

name	is the unique name assigned to the component.

```
5     : AbstractComponent(name, "Position") {
6
7     //Determines the position of the entity as a whole
8         //Currently used by the sprite system to determine the position
9     this->setAttribute_float(ATTRIBUTE_POSITION_X, 0.0);
10     this->setAttribute_float(ATTRIBUTE_POSITION_Y, 0.0);
11
12     //Determines the offset of the position. This is particularly useful
13         //for providing a view offset
14     this->setAttribute_float(ATTRIBUTE_OFFSET_X, 0.0);
15     this->setAttribute_float(ATTRIBUTE_OFFSET_Y, 0.0);
16
17     //Determines the overall objects rotation
18         //Given in radians
19     this->setAttribute_float(ATTRIBUTE_ROTATION, 0.0);
20 }
```

The documentation for this class was generated from the following files:

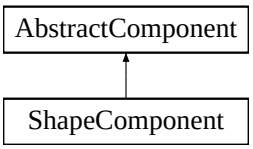
- PositionComponent.h
- PositionComponent.cpp

4.22 ShapeComponent Class Reference

Is used to express the entity as a shape on the screen.

`#include <ShapeComponent.h>`

Inheritance diagram for ShapeComponent:



Public Member Functions

- ShapeComponent (std::string name)

  The constructor for shape component.
- int update ()

  updates the shape component when modifications are made.

4.22.1 Detailed Description

Is used to express the entity as a shape on the screen.

The shape component can be used to show the entity on the screen as the expressed shape as outlined in this shape components attributes.

The shape component has also been fashioned to produce polygons.

4.22.2 Constructor & Destructor Documentation

4.22.2.1 ShapeComponent::ShapeComponent (std::string *name*)

The constructor for shape component.

The shape component has overriden the family by using "Shape." New components that inherit from the abstract component should not consider using this family name.

Parameters

name	is the unique name assigned to a created component.

```
8                                       :
9     AbstractComponent(name, "Shape") {
10     std::string keyname = name + std::to_string(reinterpret_cast<int>(this));
11     //unique keyname for our referenced text object
12
13     auto shapeManager = ShapeManager::getInstance();
14     //we don't know what type of shape it's going to be, so to simplify,
15     // just going to create one of each and decide from there
16     shapeManager->addCircleShape(keyname, std::shared_ptr<sf::CircleShape> (new sf::CircleShape()));
17     shapeManager->addRectangleShape(keyname, std::shared_ptr<sf::RectangleShape> (new sf::RectangleShape())
       );
18     shapeManager->addConvexShape(keyname, std::shared_ptr<sf::ConvexShape> (new sf::ConvexShape()));
19
20     //**********************************
21     //********** ATTRIBUTES *************
22     //**********************************
23
24     //the keyname is stored as an attribute for later retrieval
25     setAttribute_string(ATTRIBUTE_KEYNAME, keyname);
26
27     //Determines whether the given shape should be shown
28         //0 - disable
29         //1 - enable
30     setAttribute_int(ATTRIBUTE_ENABLE, 1);
31
32     //The shape type
33     setAttribute_string(ATTRIBUTE_SHAPE_TYPE,
       SHAPETYPE_CIRCLE);
34
```

```
35      //The shape attributes
36      setAttribute_float(ATTRIBUTE_RADIUS, 0.0);
37      setAttribute_float(ATTRIBUTE_WIDTH, 0.0);
38      setAttribute_float(ATTRIBUTE_HEIGHT, 0.0);
39      setAttribute_floatArray(ATTRIBUTE_POLYGON_POINTS,
      componentFloatArrayType());
40
41      //The shape fill
42      setAttribute_int(ATTRIBUTE_FILL_COLOR_RED, 255);
43      setAttribute_int(ATTRIBUTE_FILL_COLOR_GREEN, 255);
44      setAttribute_int(ATTRIBUTE_FILL_COLOR_BLUE, 255);
45      setAttribute_int(ATTRIBUTE_FILL_COLOR_ALPHA, 255);
46
47      //The shape outline
48      setAttribute_int(ATTRIBUTE_OUTLINE_COLOR_RED, 255);
49      setAttribute_int(ATTRIBUTE_OUTLINE_COLOR_GREEN, 255);
50      setAttribute_int(ATTRIBUTE_OUTLINE_COLOR_BLUE, 255);
51      setAttribute_int(ATTRIBUTE_OUTLINE_COLOR_ALPHA, 255);
52
53      //The shape outline thickness
54      setAttribute_float(ATTRIBUTE_OUTLINE_THICKNESS, 1.0);
55
56      //Related Sprite attributes
57      //This provides the offset of the sprite from its origin within the X and Y direction
58          //Useful for a multisprite entity, you provide an offset to change its placement
59      this->setAttribute_float(ATTRIBUTE_OFFSET_X, 0.0);
60      this->setAttribute_float(ATTRIBUTE_OFFSET_Y, 0.0);
61
62      //This provides the origin, or center point where rotation and the point of positioning is determined
63      this->setAttribute_float(ATTRIBUTE_ORIGIN_X, 0.0);
64      this->setAttribute_float(ATTRIBUTE_ORIGIN_Y, 0.0);
65
66      //Provides the depth of the sprite, lower means farther away, which means it
67          //will get covered by anything with a higher z-buffer {ex A(0.1) covers B(0.0)}
68      this->setAttribute_float(ATTRIBUTE_ZBUFFER, 1.0);
69
70      //The scale of the sprite being used
71          //relative to the position component
72      this->setAttribute_float(ATTRIBUTE_SCALE_X, 1.0);
73      this->setAttribute_float(ATTRIBUTE_SCALE_Y, 1.0);
74
75      //Determines the rotation of the sprite
76          //relative to a given position component
77          //given in radians
78      this->setAttribute_float(ATTRIBUTE_ROTATION, 0.0);
79
80      //perform an update on our component to form the default shape instance
81      this->update();
}
```

4.22.3 Member Function Documentation

4.22.3.1 int ShapeComponent::update ( ) [virtual]

updates the shape component when modifications are made.

It is important to perform updates on the shape component after any modifications are made during execution. Otherwise the effects will not be displayed on the screen.

Returns

Returns a zero value if the update was successful.

Implements AbstractComponent.

```
83                          {
84      auto shapeManager = ShapeManager::getInstance();
85
86      auto shapeType = this->getAttribute_string(
      ATTRIBUTE_SHAPE_TYPE);
87      if (shapeType == SHAPETYPE_CIRCLE) {
88          auto theShape = shapeManager->getCircleShape(getAttribute_string(
      ATTRIBUTE_KEYNAME));
89
90          //set the radius
91          theShape->setRadius(getAttribute_float(
      ATTRIBUTE_RADIUS));
92
93          //perform actions which are relevant to both the circle and shape
```

```
94          //get the attributes related to the outline color and apply it to the shape
95          theShape->setOutlineColor(sf::Color(
96              getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_RED),
97              getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_GREEN),
98              getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_BLUE),
99              getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_ALPHA)
100             ));
101
102         //get attribute related to the outline thickness and apply to the shape
103         float lineThickness = getAttribute_float(
    ATTRIBUTE_OUTLINE_THICKNESS);
104         theShape->setOutlineThickness(lineThickness);
105
106         //get attributes related to the fill color and apply it to the shape
107         theShape->setFillColor(sf::Color(
108             getAttribute_int(ATTRIBUTE_FILL_COLOR_RED),
109             getAttribute_int(ATTRIBUTE_FILL_COLOR_GREEN),
110             getAttribute_int(ATTRIBUTE_FILL_COLOR_BLUE),
111             getAttribute_int(ATTRIBUTE_FILL_COLOR_ALPHA)));
112
113     }
114     else if (shapeType == SHAPETYPE_RECTANGLE) {
115         auto theShape = shapeManager->getRectangleShape(getAttribute_string(
    ATTRIBUTE_KEYNAME));
116
117         //set the width and height
118         float rectWidth = getAttribute_float(ATTRIBUTE_WIDTH);
119         float rectHeight = getAttribute_float(ATTRIBUTE_HEIGHT);
120         theShape->setSize(sf::Vector2f(rectWidth, rectHeight));
121
122         //perform actions which are relevant to both the circle and shape
123         //get the attributes related to the outline color and apply it to the shape
124         theShape->setOutlineColor(sf::Color(
125             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_RED),
126             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_GREEN),
127             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_BLUE),
128             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_ALPHA)
129             ));
130
131         //get attribute related to the outline thickness and apply to the shape
132         float lineThickness = getAttribute_float(
    ATTRIBUTE_OUTLINE_THICKNESS);
133         theShape->setOutlineThickness(lineThickness);
134
135         //get attributes related to the fill color and apply it to the shape
136         theShape->setFillColor(sf::Color(
137             getAttribute_int(ATTRIBUTE_FILL_COLOR_RED),
138             getAttribute_int(ATTRIBUTE_FILL_COLOR_GREEN),
139             getAttribute_int(ATTRIBUTE_FILL_COLOR_BLUE),
140             getAttribute_int(ATTRIBUTE_FILL_COLOR_ALPHA)));
141     }
142     else if (shapeType == SHAPETYPE_POLYGON) {
143         auto theShape = shapeManager->getConvexShape(getAttribute_string(
    ATTRIBUTE_KEYNAME));
144
145         //find the polygon count
146         auto polygonPoints = *this->getAttribute_floatArray(
    ATTRIBUTE_POLYGON_POINTS);
147         assert((polygonPoints.size() % 2 == 0) && "Odd number of coordinates to describe polygon, must be
     even");
148         int numPoints = polygonPoints.size() / 2;
149
150         //set the number of points
151         theShape->setPointCount(numPoints);
152
153         //set the coordinates for each point
154         for (int i = 0; i < numPoints; i++) {
155             theShape->setPoint(i, sf::Vector2f(polygonPoints[i*2], polygonPoints[i*2+1]));
156         }
157
158         //perform actions which are relevant to both the circle and shape
159         //get the attributes related to the outline color and apply it to the shape
160         theShape->setOutlineColor(sf::Color(
161             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_RED),
162             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_GREEN),
163             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_BLUE),
164             getAttribute_int(ATTRIBUTE_OUTLINE_COLOR_ALPHA)
165             ));
166
167         //get attribute related to the outline thickness and apply to the shape
168         float lineThickness = getAttribute_float(
    ATTRIBUTE_OUTLINE_THICKNESS);
169         theShape->setOutlineThickness(lineThickness);
170
171         //get attributes related to the fill color and apply it to the shape
172         theShape->setFillColor(sf::Color(
173             getAttribute_int(ATTRIBUTE_FILL_COLOR_RED),
```

```
174             getAttribute_int(ATTRIBUTE_FILL_COLOR_GREEN),
175             getAttribute_int(ATTRIBUTE_FILL_COLOR_BLUE),
176             getAttribute_int(ATTRIBUTE_FILL_COLOR_ALPHA)));
177     }
178     else {
179         assert(0 && "Not shape by that type");
180     }
181
182
183     return 0;
184 }
```

The documentation for this class was generated from the following files:

- ShapeComponent.h
- ShapeComponent.cpp

4.23 ShapeManager Class Reference

Used to add / store, and then retrieve shapes for shape components.

`#include <ShapeManager.h>`

Public Member Functions

- void addCircleShape (const std::string &name, std::shared_ptr< sf::CircleShape > cshape)

  Add circle shape to the shape manager.
- void removeCircleShape (const std::string &name)

  Remove circle shape from the shape manager.
- bool hasCircleShape (const std::string &name)

  Check if circle shape exists within the shape manager.
- std::shared_ptr< sf::CircleShape > getCircleShape (const std::string &name)

  Get the circle shape by the given key name.
- void addRectangleShape (const std::string &name, std::shared_ptr< sf::RectangleShape > rshape)

  Add rectangle shape to the shape manager.
- void removeRectangleShape (const std::string &name)

  Remove rectangle shape from the shape manager.
- bool hasRectangleShape (const std::string &name)

  Check if rectangle shape exists within the shape manager.
- std::shared_ptr
  < sf::RectangleShape > getRectangleShape (const std::string &name)

  Get the rectangle shape by the given key name.
- void addConvexShape (const std::string &name, std::shared_ptr< sf::ConvexShape > shape)

  Add convex polygon to the shape manager.
- void removeConvexShape (const std::string &name)

  Remove convex polygon from the shape manager.
- bool hasConvexShape (const std::string &name)

  Check if convex polygon exits with the shape manager.
- std::shared_ptr< sf::ConvexShape > getConvexShape (const std::string &name)

  Get the convex polygon with the given key name.

Static Public Member Functions

- static ShapeManager * getInstance ()

  Singleton method to return the shape manager instance.

4.23.1 Detailed Description

Used to add / store, and then retrieve shapes for shape components.

The shape manager is implemented as a singleton, which manages the shapes being used by the shape components representing certain entities on the screen.

4.23.2 Member Function Documentation

4.23.2.1 static ShapeManager* ShapeManager::getInstance ( ) [inline], [static]

Singleton method to return the shape manager instance.

Returns

The shape manager instance

```
74                                      {
75     static ShapeManager _instance;
76     return &_instance;
77   }
```

The documentation for this class was generated from the following files:

- ShapeManager.h
- ShapeManager.cpp

4.24 SingletonT< InstanceClass > Class Template Reference

Static Public Member Functions

- static std::shared_ptr
  < InstanceClass > **getInstance** ()

Static Protected Attributes

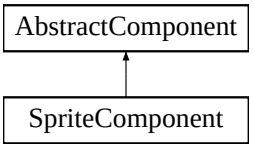
- static std::shared_ptr
  < InstanceClass > **_instance**

The documentation for this class was generated from the following file:

- SingletonT.h

4.25 SpriteComponent Class Reference

Inheritance diagram for SpriteComponent:



Public Member Functions

- **SpriteComponent** (std::string)
- int update ()

  Updates the component based on the current set of attributes.

4.25.1 Member Function Documentation

4.25.1.1 int SpriteComponent::update ( ) [virtual]

Updates the component based on the current set of attributes.

Used to update the component. This should be performed after modifications have been made on the component's attributes or members.

Returns

Returns a non-zero value if it is successful.

Implements AbstractComponent.

```
39                              {
40     return 0;
41 }
```

The documentation for this class was generated from the following files:

- SpriteComponent.h
- SpriteComponent.cpp

4.26 SpriteManager Class Reference

Used to manage the sprites for sprite components.

`#include <SpriteManager.h>`

Public Member Functions

- void addSprite (const std::string &name, std::shared_ptr< spriteType > theSprite)

  Adds a sprite to the sprite manager.
- void removeSprite (const std::string &name)

  Removes a sprite from the sprite manager.
- bool hasSprite (const std::string &name)

  Checks if the sprite by the given name exists.
- std::shared_ptr< spriteType > getSprite (const std::string &name)

  Gets the sprite by the given unique name.

Static Public Member Functions

- static SpriteManager * getInstance ()

  Singleton class method to get the single instance.

4.26.1 Detailed Description

Used to manage the sprites for sprite components.

Sprite manager is a class which manages the sprites that are to be used within the game.

4.26.2 Member Function Documentation

4.26.2.1 void SpriteManager::addSprite (const std::string & *name*, std::shared_ptr< spriteType > *theSprite*)

Adds a sprite to the sprite manager.

Parameters

name	is the unique name that is assigned to the sprite
theSprite	is the sprite to assign to the manager.

```
9                                                                                {
10      assert(!this->hasSprite(name) && "Sprite by that name already exists");
11      this->spriteList[name] = theSprite;
12  }
```

4.26.2.2 static SpriteManager* SpriteManager::getInstance ( ) [inline], [static]

Singleton class method to get the single instance.

Returns

Returns the sprite manager single instance.

```
99                                      {
100     static SpriteManager _instance;
101     return &_instance;
102   }
```

4.26.2.3 std::shared_ptr< spriteType > SpriteManager::getSprite (const std::string & *name*)

Gets the sprite by the given unique name.

Parameters

name	is the unique name that is assigned to the sprite

Returns

Returns the shared_ptr referencing the sprite.

TODO: exception handling.

```
27                                                                          {
28      assert(this->hasSprite(name) && "Sprite doesn't exist by that name");
29      return this->spriteList[name];
30  }
```

4.26.2.4 bool SpriteManager::hasSprite (const std::string & *name*)

Checks if the sprite by the given name exists.

Parameters

name	is the unique name that is assigned to the sprite

Returns

Returns true if the given sprite exists, otherwise it returns false.

```
20                                                  {
21     if(this->spriteList.find(name) == this->spriteList.end()) {
22         return false;
23     }
24     return true;
25 }
```

4.26.2.5 void SpriteManager::removeSprite (const std::string & *name*)

Removes a sprite from the sprite manager.

Parameters

name	is the unique name that is assigned to the sprite

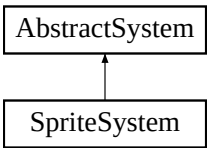
```
14                                                  {
15     assert(this->hasSprite(name) && "Sprite doesn't exist");
16     //remove element by key
17     this->spriteList.erase(name);
18 }
```

The documentation for this class was generated from the following files:

- SpriteManager.h
- SpriteManager.cpp

4.27 SpriteSystem Class Reference

Inheritance diagram for SpriteSystem:



Public Member Functions

- **SpriteSystem** (sf::RenderWindow &)
- int process ()

  Perform actions on entities.

Additional Inherited Members

4.27.1 Member Function Documentation

4.27.1.1 int SpriteSystem::process ( ) [virtual]

Perform actions on entities.

The process method is a pure virtual function that processes all of th entities stored within the entity manager. Some systems do not use the entity manager for processing, but may process other things that require processing per frame.

What is processed is fully dependant on the implementation.

Returns

A non-zero value if the processing was successful.

Implements AbstractSystem.

```
26                          {
27      //first we need to grab the list of entities
28      //for now, we're going to grab all of the entities
29      auto entityVector = this->entityManager->getAllEntities();
30      std::list<std::shared_ptr<entityType>> entityList;
31      for (auto entityValue : entityVector) {
32          entityList.push_back(entityValue);
33      }
34      //sort the entities based on it's Z-buffer. The sorting criteria is based on the minimum Z-buffer found
35      // within the entities
36
37
38      for (auto entity : entityList) {
39          //determine if the entity has the desired components to work with
40          if ((!entity->hasComponentFamily("Sprite") ||
41              !entity->hasComponentFamily("Position")) &&
42              (!entity->hasComponentFamily("Shape") ||
43              !entity->hasComponentFamily("Position")) &&
44              (!entity->hasComponentFamily("Text") ||
45              !entity->hasComponentFamily("Position"))) {
46              continue;
47          }
48
49          //grab all of the desired components
50          auto theSpriteComponents = entity->getComponentsByFamily("Sprite");
51          auto theShapeComponents = entity->getComponentsByFamily("Shape");
52          auto theTextComponents = entity->getComponentsByFamily("Text");
53          auto thePositionComponents = entity->getComponentsByFamily("Position");
54
55          //merge the shape, sprite and text components within the same list
56          std::list<std::shared_ptr<componentType>> drawableComponents;
57          for (auto sprite : theSpriteComponents) {
58              drawableComponents.push_back(sprite);
59          }
60          for (auto shape : theShapeComponents) {
61              drawableComponents.push_back(shape);
62          }
63
64          for (auto text : theTextComponents) {
65              drawableComponents.push_back(text);
66          }
67
68          //we need to sort the drawables based on the provided Z-Buffer Attribute
69          drawableComponents.sort([] (std::shared_ptr<componentType> first, std::shared_ptr<componentType>
    second) {
70              float zFirst = first->getAttribute_float(ATTRIBUTE_ZBUFFER);
71              float zSecond = second->getAttribute_float(ATTRIBUTE_ZBUFFER);
72              return (zFirst < zSecond);
73          });
74
75          //we are going limit the scope of entities to one position component for now
76          assert(thePositionComponents.size() == 1 && "Entities are limited to one position component");
77
78          //now we can just assume that we have only one position component
79          auto positionComponent = thePositionComponents.front();
80
81          //grab and store the position and rotation for our entities
82          float positionX = positionComponent->getAttribute_float(
    ATTRIBUTE_POSITION_X);
83          float positionY = positionComponent->getAttribute_float(
    ATTRIBUTE_POSITION_Y);
84
85          float rotation = positionComponent->getAttribute_float(
    ATTRIBUTE_ROTATION);
86
```

```
87          //next, we need to process each sprite component and offset it
88          //by the position component
89          //TODO: sort by Z-buffer
90          for (auto theComponent : drawableComponents) {
91              //first, we determine if the sprite is enabled to draw
92              int bEnabled = theComponent->getAttribute_int(ATTRIBUTE_ENABLE);
93              if (!bEnabled) {
94                  continue;
95              }
96
97              //Intialize our drawable ptr object
98              std::shared_ptr<sf::Drawable> theDrawable;
99              std::shared_ptr<sf::Transformable> theTransform;
100              std::shared_ptr<sf::Shape> theShape;
101              std::shared_ptr<spriteType> theSprite;
102              std::shared_ptr<sf::CircleShape> theCircle;
103              std::shared_ptr<sf::RectangleShape> theRectangle;
104              std::shared_ptr<sf::ConvexShape> theConvexShape;
105              std::shared_ptr<sf::Text> theText;
106
107              //get our sprite from the shape component, or use the default
108              if (theComponent->getFamily() == "Sprite") {
109                  bool bDefault = true;
110                  //get the name of the sprite key name being stored in the manager
111                  std::string spriteName = theComponent->getAttribute_string("SpriteName");
112                  //Check if the sprite being used is the default
113                  if (spriteName != "None") {
114                      bDefault = false;
115                  }
116
117                  //if it's a default, use the default sprite image
118                  if (bDefault) {
119                      theSprite = this->spriteManager->getSprite(
    DEFAULT_SPRITE);
120                  }
121                  //else, grab our sprite TODO: checking
122                  else { //bDefault == false
123                      theSprite = this->spriteManager->getSprite(spriteName);
124                  }
125
126                  //cast to a drawable and a transformable
127                  theTransform = std::static_pointer_cast<sf::Transformable> (theSprite);
128                  assert(theTransform != nullptr && "Nullptr in transform");
129                  theDrawable = std::static_pointer_cast<sf::Drawable> (theSprite);
130                  assert(theDrawable != nullptr && "Nullptr in drawable");
131              } //END if (theComponent->getFamily() == "Sprite") { ...
132              //figure out the shape to place within our sprite
133              else if (theComponent->getFamily() == "Shape") {
134                  //intialize the shape base class
135                  std::string shapeType = theComponent->getAttribute_string(
    ATTRIBUTE_SHAPE_TYPE);
136                  if (shapeType == SHAPETYPE_CIRCLE) {
137                      theCircle = shapeManager->getCircleShape(theComponent->
    getAttribute_string(ATTRIBUTE_KEYNAME));
138                      theShape = std::static_pointer_cast<sf::Shape> (theCircle);
139                      assert(theShape != nullptr && "Nullptr in circle shape");
140                  }
141                  else if (shapeType == SHAPETYPE_RECTANGLE) {
142                      theRectangle = shapeManager->getRectangleShape(theComponent->
    getAttribute_string(ATTRIBUTE_KEYNAME));
143                      //cast it to the base class sf::Shape
144                      theShape = std::static_pointer_cast<sf::Shape> (theRectangle);
145                      assert(theShape != nullptr && "Nullptr in rectangle shape");
146                  }
147                  else if (shapeType == SHAPETYPE_POLYGON) {
148                      theConvexShape = shapeManager->getConvexShape(theComponent->
    getAttribute_string(ATTRIBUTE_KEYNAME));
149                      //cast to the base class sf::Shape
150                      theShape = std::static_pointer_cast<sf::Shape> (theConvexShape);
151                      assert(theShape != nullptr && "Nullptr in convex shape");
152                  }
153                  else {
154                      assert(0 && "Shape Type does not exist");
155                  }
156
157                  //we cast to pointers representing transformable
158                  //  and drawable objects in order to relate to the other given sprite
159                  //  class. This allows it to be transformed like the sprite
160                  theTransform = std::static_pointer_cast<sf::Transformable> (theShape);
161                  assert(theTransform != nullptr && "Nullptr in shape transform");
162                  theDrawable = std::static_pointer_cast<sf::Drawable> (theShape);
163                  assert(theDrawable != nullptr && "Nullptr in shape drawable");
164              } //END if (theComponent->getFamily() == "Shape") { ...
165              else if (theComponent->getFamily() == "Text") {
166                  theText = textManager->getText(theComponent->getAttribute_string(
    ATTRIBUTE_KEYNAME));
167
```

```
168                 //we cast to pointers representing transformable
169                 //  and drawable objects in order to relate to the other given sprite
170                 //  class. This allows it to be transformed like the sprite
171                 theTransform = std::static_pointer_cast<sf::Transformable> (theText);
172                 assert(theTransform != nullptr && "Nullptr in shape transform");
173                 theDrawable = std::static_pointer_cast<sf::Drawable> (theText);
174                 assert(theDrawable != nullptr && "Nullptr in shape drawable");
175             }//END else if (theComponent->getFamily() == "Text") {
176 
177             //grab the rest of the desired attributes
178 
179             //get the transformable offset, which is
180             //  relative to the position component and its offset
181             float offsetX = theComponent->getAttribute_float(ATTRIBUTE_OFFSET_X) +
182                 positionComponent->getAttribute_float(ATTRIBUTE_OFFSET_X);
183             float offsetY = theComponent->getAttribute_float(ATTRIBUTE_OFFSET_Y) +
184                 positionComponent->getAttribute_float(ATTRIBUTE_OFFSET_Y);
185 
186             //get the transformable scale
187             float scaleX = theComponent->getAttribute_float(ATTRIBUTE_SCALE_X);
188             float scaleY = theComponent->getAttribute_float(ATTRIBUTE_SCALE_Y);
189 
190             //get the transformable origin point
191             float originX = theComponent->getAttribute_float(ATTRIBUTE_ORIGIN_X);
192             float originY = theComponent->getAttribute_float(ATTRIBUTE_ORIGIN_Y);
193 
194             //get the rotation of the transformable relative to the position
195             float spriteRotation = theComponent->getAttribute_float(
    ATTRIBUTE_ROTATION);
196 
197             //PERFORM TRANSFORM OPERATIONS
198 
199             //First we set the origin of our transformable object
200             theTransform->setOrigin(originX, originY);
201 
202             //SFML rotation is presented in degrees, converting from radians
203             float rot = ((rotation + spriteRotation) / (2.f * PI)) * 360.f;
204 
205             //fix the angle between 0 and 360 degrees
206             if (rot >= 360) {
207                 int numRotations = (int)rot / 360;
208                 rot = rot - 360 * numRotations;
209             }
210 
211             //given situations where it is close to zero, set it to zero
212             if ((int)rot % 360 == 0){
213                 if ((rot - floor(rot)) < FULL_ROTATION_THRESHOLD) {
214                     theComponent->setAttribute_float(ATTRIBUTE_ROTATION, 0.);
215                     positionComponent->setAttribute_float(ATTRIBUTE_ROTATION, 0.);
216                     rot = 0;
217                 }
218             }
219             theTransform->setRotation(rot);
220 
221             //Scale the sprite
222             theTransform->setScale(scaleX, scaleY);
223 
224             //move the sprite
225             //we need to correct for the changed origin
226             theTransform->setPosition(
227                 positionX + offsetX,
228                 positionY + offsetY
229                 );
230 
231             //Finally, draw our sprite
232             window.draw(*theDrawable);
233 
234         } //END for (auto theComponent : ...
235     } //END for (auto entity : ...
236     return 0;
237 }
```

The documentation for this class was generated from the following files:

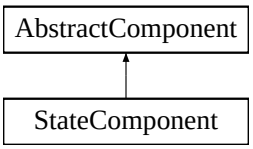
- SpriteSystem.h
- SpriteSystem.cpp

4.28 StateComponent Class Reference

Used to store entity state (Not currently used)

`#include <StateComponent.h>`

Inheritance diagram for StateComponent:



Public Member Functions

- StateComponent (std::string name)

  Constructor overrides the AbstractComponent family to be "State".
- int update ()

  Used when the component is modified (Not Implemented)

4.28.1 Detailed Description

Used to store entity state (Not currently used)

4.28.2 Constructor & Destructor Documentation

4.28.2.1 StateComponent::StateComponent (std::string *name*)

Constructor overrides the AbstractComponent family to be "State".

Parameters

name	is the unique name assigned to the component.

```
5                                          :
6     AbstractComponent(name, COMPONENT_FAMILY_STATE) {
7
8     //Determines whether the given state is active, and should
9         //trigger the given callback
10     this->setAttribute_int(ATTRIBUTE_ENABLE, 1);
11
12     //Key of the callback function to trigger
13     this->setAttribute_string(ATTRIBUTE_CALLBACK, "None");
14
15     //determines if the given state callback will be repeated
16     this->setAttribute_int("bRepeat", 1);
}
```

4.28.3 Member Function Documentation

4.28.3.1 int StateComponent::update ( ) [virtual]

Used when the component is modified (Not Implemented)

Returns

Returns a zero value if it was successful.

Implements AbstractComponent.

```
18                          {
19     return 0;
20 }
```

The documentation for this class was generated from the following files:

- StateComponent.h
- StateComponent.cpp

4.29 StateManager Class Reference

Public Member Functions

- void **addStateCallback** (const std::string &, functionTemplate)
- void **setEnable** (const std::string &)
- void **setDisable** (const std::string &)
- stateType **getState** (const std::string &)
- bool **hasState** (const std::string &)

Static Public Member Functions

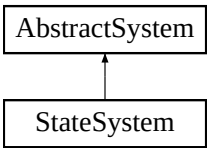
- static StateManager * **getInstance** ()

The documentation for this class was generated from the following files:

- StateManager.h
- StateManager.cpp

4.30 StateSystem Class Reference

Inheritance diagram for StateSystem:



Public Member Functions

- void **setStateMachineFunction** (functionStateMachineTemplate)
- int **triggerStateMachine** ()
- int process ()

  Perform actions on entities.

Additional Inherited Members

4.30.1 Member Function Documentation

4.30.1.1 int StateSystem::process ( ) [virtual]

Perform actions on entities.

The process method is a pure virtual function that processes all of th entities stored within the entity manager. Some systems do not use the entity manager for processing, but may process other things that require processing per frame.

What is processed is fully dependant on the implementation.

Returns

A non-zero value if the processing was successful.

Implements AbstractSystem.

```
25                          {
26     //run the state machine function
27     if (this->stateMachineFunction()) {
28         assert(0 && "State machine returned a non-zero value");
29     }
30 
31     //grab the state components from the entities
32     for (auto entity : this->entityManager->getAllEntities()) {
33         //get only the state components
34         for (auto stateComponent : entity->getComponentsByFamily(
       COMPONENT_FAMILY_STATE)) {
35             //weed out the components that aren't enabled
36             if (!stateComponent->getAttribute_int(ATTRIBUTE_ENABLE)) {
37                 continue;
38             }
39 
40             //perform the state component callback
41             auto callbackString = stateComponent->getAttribute_string(
       ATTRIBUTE_CALLBACK);
42             callbackManager->triggerCallback(callbackString, entity->getID());
43 
44             //check to see if it's on repeat, otherwise disable the state component
45             // from executing again
46             if (!stateComponent->getAttribute_int("bRepeat")) {
47                 stateComponent->setAttribute_int(ATTRIBUTE_ENABLE, 0);
48             }
49 
50         } //END for (auto stateComponent : ...
51     } //END for (auto entity : ...
52     return 0;
53 }
```

The documentation for this class was generated from the following files:

- StateSystem.h
- StateSystem.cpp

4.31 SystemManager Class Reference

Public Member Functions

- void **addSystem** (std::shared_ptr< systemType >)
- bool **hasSystem** (const std::string &)
- void **removeSystem** (const std::string &)
- int **processSystemList** ()

Static Public Member Functions

- static SystemManager * **getInstance** ()

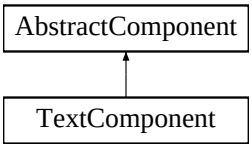
The documentation for this class was generated from the following files:

- SystemManager.h
- SystemManager.cpp

4.32 TextComponent Class Reference

`#include <TextComponent.h>`

Inheritance diagram for TextComponent:



Public Member Functions

- TextComponent (std::string name)

  Text component constructor.
- int update ()

  Updates the component to include any modifications.

4.32.1 Detailed Description

The text component inherits the abstract component, and is used to show text on the screen. Different attributes assigned change how the text is displayed.

4.32.2 Constructor & Destructor Documentation

4.32.2.1 TextComponent::TextComponent (std::string *name*)

Text component constructor.

The text component overrides the abstract component family to be "Text"

Parameters

name	is the unique name assigned to the given text component.

```
9                                       :
10      AbstractComponent(name, "Text") {
11
12      std::string keyname = name + std::to_string(reinterpret_cast<int>(this));
13      //unique keyname for our referenced text object
14
15      //need to create our text object and place it within the text manager
16      auto textManager = TextManager::getInstance();
17      textManager->addText(keyname, std::shared_ptr<sf::Text> (new sf::Text()));
18
19
20      //***********************************
21      //********** ATTRIBUTES *************
22      //***********************************
23
24      //the keyname is stored as an attribute for later retrieval
25      setAttribute_string(ATTRIBUTE_KEYNAME, keyname);
26
27      //Determines whether the given shape should be shown
28          //0 - disable
29          //1 - enable
30      setAttribute_int(ATTRIBUTE_ENABLE, 1);
31
32      //Text string to show on the screen
33      setAttribute_string(ATTRIBUTE_TEXT_STRING,
      DEFAULT_TEXT_STRING);
34
35      //Text string font
36      setAttribute_int(ATTRIBUTE_TEXT_SIZE, 30);
```

```
37     setAttribute_int(ATTRIBUTE_TEXT_STYLE_UNDERLINE, 0);
38     setAttribute_int(ATTRIBUTE_TEXT_STYLE_BOLD, 0);
39     setAttribute_int(ATTRIBUTE_TEXT_STYLE_ITALIC, 0);
40     setAttribute_string(ATTRIBUTE_TEXT_FONT,
       DEFAULT_FONT);
41
42     //The text fill color
43     setAttribute_int(ATTRIBUTE_FILL_COLOR_RED, 255);
44     setAttribute_int(ATTRIBUTE_FILL_COLOR_GREEN, 255);
45     setAttribute_int(ATTRIBUTE_FILL_COLOR_BLUE, 255);
46     setAttribute_int(ATTRIBUTE_FILL_COLOR_ALPHA, 255);
47
48     //Related Sprite attributes
49     //This provides the offset of the collision bound from its origin within the X and Y direction
50     this->setAttribute_float(ATTRIBUTE_OFFSET_X, 0.0);
51     this->setAttribute_float(ATTRIBUTE_OFFSET_Y, 0.0);
52
53     //This provides the origin, or center point where rotation and the point of positioning is determined
54     this->setAttribute_float(ATTRIBUTE_ORIGIN_X, 0.0);
55     this->setAttribute_float(ATTRIBUTE_ORIGIN_Y, 0.0);
56
57     //Provides the depth of the sprite, lower means farther away, which means it
58         //will be covered by anything with a higher z-buffer {ex A(0.1) covers B(0.0)}
59     this->setAttribute_float(ATTRIBUTE_ZBUFFER, 1.0);
60
61     //The scale of the sprite being used
62         //relative to the position component
63     this->setAttribute_float(ATTRIBUTE_SCALE_X, 1.0);
64     this->setAttribute_float(ATTRIBUTE_SCALE_Y, 1.0);
65
66     //Determines the rotation of the sprite
67         //relative to a given position component
68         //given in radians
69     this->setAttribute_float(ATTRIBUTE_ROTATION, 0.0);
70
71     //After all of the attributes are added, we perform an update to push
72     //  all of our defaults onto the text instance
73     this->update();
74 }
```

4.32.3 Member Function Documentation

4.32.3.1 int TextComponent::update ( ) [virtual]

Updates the component to include any modifications.

The text component update command needs to be called anytime an attribute is modified.

Returns

A zero value when it is successful.

Implements AbstractComponent.

```
76                          {
77     auto textManager = TextManager::getInstance();
78     auto theText = textManager->getText(getAttribute_string(
       ATTRIBUTE_KEYNAME));
79
80     //set the string to display
81     theText->setString(getAttribute_string(
       ATTRIBUTE_TEXT_STRING));
82
83     //set the character size
84     theText->setCharacterSize(getAttribute_int(
       ATTRIBUTE_TEXT_SIZE));
85
86     //set the text style
87     int textAttributes = (getAttribute_int(
       ATTRIBUTE_TEXT_STYLE_BOLD)) ?
88         sf::Text::Bold : 0;
89     textAttributes |= (getAttribute_int(
       ATTRIBUTE_TEXT_STYLE_ITALIC)) ?
90         sf::Text::Italic : 0;
91     textAttributes |= (getAttribute_int(
       ATTRIBUTE_TEXT_STYLE_UNDERLINE)) ?
92         sf::Text::Underlined : 0;
93     theText->setStyle(textAttributes);
94
```

```
95      //check and set our font
96      if (textManager->hasFont(getAttribute_string(
      ATTRIBUTE_TEXT_FONT))) {
97          theText->setFont(*(textManager->getFont(getAttribute_string(
      ATTRIBUTE_TEXT_FONT))));
98      }
99
100     //set our text color
101     theText->setColor(
102         sf::Color(
103             getAttribute_int(ATTRIBUTE_FILL_COLOR_RED),
104             getAttribute_int(ATTRIBUTE_FILL_COLOR_GREEN),
105             getAttribute_int(ATTRIBUTE_FILL_COLOR_BLUE),
106             getAttribute_int(ATTRIBUTE_FILL_COLOR_ALPHA))
107     );
108
109     return 0;
110 }
```

The documentation for this class was generated from the following files:

- TextComponent.h
- TextComponent.cpp

4.33 TextManager Class Reference

Used to store / add, and retrieve text for use with the text component.

`#include <TextManager.h>`

Public Member Functions

- void addText (const std::string &name, std::shared_ptr< sf::Text > text)

  Add text to the text manager.
- void removeText (const std::string &name)

  Remove text from the text manager.
- bool hasText (const std::string &name)

  Check if text exists within the manager.
- std::shared_ptr< sf::Text > getText (const std::string &name)

  Get text from the text manager.
- void addFont (const std::string &name, std::shared_ptr< sf::Font > font)

  Add font to the text manager.
- void removeFont (const std::string &name)

  Remove font from the text manager.
- bool hasFont (const std::string &name)

  Checks if font exists within the text manager.
- std::shared_ptr< sf::Font > getFont (const std::string &name)

  Get font from the text manager.

Static Public Member Functions

- static TextManager * getInstance ()

  Singleton method, retrieves the single instance.

4.33.1 Detailed Description

Used to store / add, and retrieve text for use with the text component.

The text manager is used to store and retrieve text for the text components. It also contains facilities for storing and retrieving fonts that are to be used for the text's font type.

4.33.2 Member Function Documentation

4.33.2.1 void TextManager::addFont (const std::string & *name*, std::shared_ptr< sf::Font > *font*)

Add font to the text manager.

Parameters

name	is the unique name assigned to the text object
font	is the font object to be stored within the text manager.

```
27                                                                          {
28     assert(!(this->hasFont(name)) && "Font by that name already exists");
29     this->fontMap[name] = font;
30 }
```

4.33.2.2 void TextManager::addText (const std::string & *name*, std::shared_ptr< sf::Text > *text*)

Add text to the text manager.

Parameters

name	is the unique name assigned to the text object
text	is the text object being stored within the text manager.

```
5                                                                       {
6     assert(!(this->hasText(name)) && "Text by that name already exists");
7     this->textMap[name] = text;
8 }
```

4.33.2.3 std::shared_ptr< sf::Font > TextManager::getFont (const std::string & *name*)

Get font from the text manager.

Parameters

name	is the unique name assigned to the text object

Returns

Returns the font object

TODO: Exception handling.

```
44                                                                  {
45     assert(this->hasFont(name) && "Font by that name doesn't exist");
46     return this->fontMap[name];
47 }
```

4.33.2.4 static TextManager* TextManager::getInstance ( ) [inline], [static]

Singleton method, retrieves the single instance.

Returns

Returns the text manager single instance.

```
92                                  {
93     static TextManager _instance;
94     return &_instance;
95   }
```

4.33.2.5 std::shared_ptr< sf::Text > TextManager::getText (const std::string & *name*)

Get text from the text manager.

Parameters

name	is the unique name assigned to the text object

Returns

The text object

TODO: Exception handling.

```
22                                                              {
23     assert(this->hasText(name) && "Text by that name doesn't exist");
24     return this->textMap[name];
25 }
```

4.33.2.6 bool TextManager::hasFont (const std::string & *name*)

Checks if font exists within the text manager.

Parameters

name	is the unique name assigned to the text object

Returns

Returns true if the font exists, otherwise returns false.

```
37                                              {
38     if (this->fontMap.find(name) == this->fontMap.end()) {
39         return false;
40     }
41     return true;
42 }
```

4.33.2.7 bool TextManager::hasText (const std::string & *name*)

Check if text exists within the manager.

Parameters

name	is the unique name assigned to the text object

Returns

Returns true if the text exists, otherwise it returns false.

```
15                                              {
16     if (this->textMap.find(name) == this->textMap.end()) {
17         return false;
18     }
19     return true;
20 }
```

4.33.2.8 void TextManager::removeFont (const std::string & *name*)

Remove font from the text manager.

Parameters

name	is the unique name assigned to the text object

```
32                                                  {
33     assert(this->hasFont(name) && "Font by that name doesn't exist");
34     this->fontMap.erase(name);
35 }
```

4.33.2.9 void TextManager::removeText (const std::string & *name*)

Remove text from the text manager.

Parameters

name	is the unique name assigned to the text object

```
10                                                  {
11     assert(this->hasText(name) && "Text by that name doesn't exist");
12     this->textMap.erase(name);
13 }
```

The documentation for this class was generated from the following files:

- TextManager.h
- TextManager.cpp

4.34 TextureManager Class Reference

Used to store / add and retrieve textures for the sprites.

`#include <TextureManager.h>`

Public Member Functions

- void addTexture (const std::string &name, std::shared_ptr< textureType > texture)
  Add texture to the texture manager.
- void deleteTexture (const std::string &name)
  Remove texture from the texture manager.
- bool hasTexture (const std::string &name)
  Checks if the texture exists.
- std::shared_ptr< textureType > getTexture (const std::string &name)
  Gets the texture described by the given name.

Static Public Member Functions

- static TextureManager * getInstance ()
  Singleton method, used to retrieve the single instance.

4.34.1 Detailed Description

Used to store / add and retrieve textures for the sprites.

TextureManager is a singleton

Sprites require a reference to a texture in order to display properly. This class is used to store / add and retrieve these textures.

4.34.2 Member Function Documentation

4.34.2.1 void TextureManager::addTexture (const std::string & *name*, std::shared_ptr< textureType > *texture*)

Add texture to the texture manager.

Parameters

name	is the unique name assigned to the texture
texture	is the texture object

```
9                                                                                   {
10      assert(!this->hasTexture(name) && "Texture by that name already exists");
11      this->textureMap[name] = texture;
12  }
```

4.34.2.2 void TextureManager::deleteTexture (const std::string & *name*)

Remove texture from the texture manager.

Parameters

name	is the unique name assigned to the texture

```
14                                                  {
15      assert(this->hasTexture(name) && "Texture by that name doesn't exist");
16      int numkeys = this->textureMap.erase(name);
17      assert(numkeys == 1 && "Shouldn't be more than one key");
18  }
```

4.34.2.3 static TextureManager* TextureManager::getInstance ( ) [inline], [static]

Singleton method, used to retrieve the single instance.

Returns

Returns the single instance of the texture manager.

```
76                                      {
77      static TextureManager _instance;
78      return &_instance;
79    }
```

4.34.2.4 std::shared_ptr< textureType > TextureManager::getTexture (const std::string & *name*)

Gets the texture described by the given name.

Parameters

name	is the unique name assigned to the texture

Returns

Returns true if the given texture exists, otherwise it returns false.

```
27                                                                          {
28      assert(this->hasTexture(name) && "Texture does not exist");
29      return this->textureMap[name];
30  }
```

4.34.2.5 bool TextureManager::hasTexture (const std::string & *name*)

Checks if the texture exists.

Parameters

name	is the unique name assigned to the texture

```
20                                                      {
21     if (this->textureMap.find(name) == this->textureMap.end()) {
22         return false;
23     }
24     return true;
25 }
```

The documentation for this class was generated from the following files:

- TextureManager.h
- TextureManager.cpp

Chapter 5

File Documentation

5.1 AbstractComponent.h File Reference

```
#include <map>
#include <memory>
#include <list>
#include <string>
#include <vector>
```

Classes

- union attributeContainer_union

  union container to store the attribute values
- struct attribute_struct

  structure used to represent the attribute
- class AbstractComponent

  The abstract representation for each component.

Typedefs

- typedef std::vector< int > componentIntegerArrayType

  Type to represent integer arrays in the attribute container.
- typedef std::vector< float > componentFloatArrayType

  Type to represent the float arrays in the attribute container.
- typedef union attributeContainer_union attribute_container

  union container to store the attribute values
- typedef struct attribute_struct attribute

  structure used to represent the attribute
- typedef std::map< std::string, attribute > attributeListType

  The type used to represent the list of attributes for the component.
- typedef AbstractComponent componentType

  Simple typedef to abstract the abstract...

Enumerations

- enum attribute_type {
  ATTR_INTEGER, **ATTR_FLOAT**, **ATTR_STRING**, **ATTR_FLOATARRAY**,
  ATTR_INTEGERARRAY }

  Enumeration to describe the attribute type.

5.1.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.1.2 LICENSE

5.1.3 DESCRIPTION

Contains the abstract for the components within the entity component system

5.1.4 Typedef Documentation

5.1.4.1 typedef struct attribute_struct attribute

structure used to represent the attribute

The attribute is a structure which holds both the enumeration which describes the type of value stored, and the union container which holds the value.

Parameters

attr_container	Union holding the data that represents the attribute
attr_type	The type of the data being stored for the attribute

5.1.4.2 typedef union attributeContainer_union attribute_container

union container to store the attribute values

The container holds the attribute value, and the type of value is described by the attribute_type

5.1.5 Enumeration Type Documentation

5.1.5.1 enum attribute_type

Enumeration to describe the attribute type.

Enumeration used to describe what type is being stored within the attributeContainer_union.

```
49                    {
50     ATTR_INTEGER,
51     ATTR_FLOAT,
52     ATTR_STRING,
53     ATTR_FLOATARRAY,
54     ATTR_INTEGERARRAY
55 };
```

5.2 AbstractEntity.h File Reference

```
#include <map>
#include <vector>
#include <memory>
#include "AbstractComponent.h"
```

Classes

- class AbstractEntity

  The abstract representation for each entity.

Typedefs

- typedef AbstractEntity entityType

  Typedef to abstract the abstract...
- typedef std::vector
  < std::shared_ptr
  < componentType > > componentVectorType

  The type used to store components within the entity.

5.2.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.2.2 LICENSE

5.2.3 DESCRIPTION

Contains the abstract for the entities within the entity component system

5.3 AbstractSystem.h File Reference

```
#include <memory>
#include <list>
#include <string>
#include "Entities.h"
#include "EntityManager.h"
```

Classes

- class AbstractSystem

  The abstract representation for each system.

Typedefs

- typedef AbstractSystem systemType

  Typedef to abstract the abstract...

5.3.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.3.2 LICENSE

5.3.3 DESCRIPTION

Contains the abstract for the systems within the entity component system

5.4 AE_Attributes.h File Reference

```
#include "SFML\Graphics\Color.hpp"
```

Macros

- #define ATTRIBUTE_KEYNAME "KeyName"

  Represents a unique keyname.
- #define ATTRIBUTE_CREATED "bCreated"

  Represents whether it has been created.
- #define ATTRIBUTE_ENABLE "bEnabled"

  Determines whether the given component is enabled.
- #define ATTRIBUTE_CALLBACK "Callback"

  Contains the name of the callback.
- #define ATTRIBUTE_ZBUFFER "Z-Buffer"

  Represents depth.
- #define ATTRIBUTE_SHAPE_TYPE "Shape Type"

  Represents the shape type to use.
- #define SHAPETYPE_CIRCLE "Circle"

  Circle Shape Type.
- #define SHAPETYPE_RECTANGLE "Rect"

  Rectangle Shape Type.
- #define SHAPETYPE_POLYGON "Polygon"

  Polygon Shape Type.
- #define ATTRIBUTE_RADIUS "Radius"

  the radius, used with circle shapes
- #define ATTRIBUTE_WIDTH "Width"

  the width, used with rectangle shapes
- #define ATTRIBUTE_HEIGHT "Height"

  the height, used with rectangle shapes
- #define ATTRIBUTE_POLYGON_POINTS "Attribute_Polygon_Points"

  Attribute holding polygon points.
- #define ATTRIBUTE_FILL_COLOR_RED "Fill Color Red"

  defines the fill color for red
- #define ATTRIBUTE_FILL_COLOR_GREEN "Fill Color Green"

  defines the fill color for green
- #define ATTRIBUTE_FILL_COLOR_BLUE "Fill Color Blue"

  defines the fill color for blue
- #define ATTRIBUTE_FILL_COLOR_ALPHA "Fill Color Alpha"

  defines the fill color for alpha transparency
- #define ATTRIBUTE_OUTLINE_COLOR_RED "Outline Color Red"

  defines the outline color for red
- #define ATTRIBUTE_OUTLINE_COLOR_GREEN "Outline Color Green"

  defines the outline color for green
- #define ATTRIBUTE_OUTLINE_COLOR_BLUE "Outline Color Blue"

  defines the outline color for blue
- #define ATTRIBUTE_OUTLINE_COLOR_ALPHA "Outline Color Alpha"

  defines the outline color for alpha transparency
- #define ATTRIBUTE_OUTLINE_THICKNESS "Outline Thickness"

  defines the outline thickness
- #define ATTRIBUTE_POSITION_X "Position_X"

  position attribute X
- #define ATTRIBUTE_POSITION_Y "Position_Y"

  position attribute Y
- #define ATTRIBUTE_ROTATION "Rotation"

position, sprite, and shape rotation

- #define ATTRIBUTE_SPRITE_NAME "SpriteName"

  Represents the sprite name.
- #define ATTRIBUTE_OFFSET_X "Offset_X"

  Offset of the component from the position component X.
- #define ATTRIBUTE_OFFSET_Y "Offset_Y"

  Offset of the component from the position component Y.
- #define ATTRIBUTE_ORIGIN_X "Origin_X"

  Represents the center origin for X.
- #define ATTRIBUTE_ORIGIN_Y "Origin_Y"

  Represents the center origin for X.
- #define ATTRIBUTE_SCALE_X "Scale_X"

  determines the scale of the the sprite, or shape X
- #define ATTRIBUTE_SCALE_Y "Scale_Y"

  determines the scale of the the sprite, or shape Y
- #define COLLISION_BOUND_CIRCLE "Collision_Bound_Circle"

  Represents the collision bound of a circle.
- #define COLLISION_BOUND_RECTANGLE "Collision_Bound_Rectangle"

  Represents the collision bound of a rectangle.
- #define COLLISION_BOUND_POLYGON "Collision_Bound_Polygon"

  Represents the collision bound of a polygon.
- #define ATTRIBUTE_COLLISION_BOUND_TYPE "Collision_Bound_Type"

  The attribute for representing the collision bound.
- #define ATTRIBUTE_COLLISION_TAG "Collision_Tag"

  Collision Tag for representing the given entity.
- #define COLLISION_DEFAULT_TAG "Default"

  The default collision tag.
- #define ATTRIBUTE_TEXT_STRING "Text_String"

  Attribute which holds the raw string.
- #define DEFAULT_TEXT_STRING "Default Text"

  Represents the default raw string if none are assigned.
- #define ATTRIBUTE_TEXT_SIZE "Text_Size"

  Attribute for assigning the text size.
- #define ATTRIBUTE_TEXT_STYLE_UNDERLINE "Text_Style_Underline"

  Attribute to enable or disable underline.
- #define ATTRIBUTE_TEXT_STYLE_BOLD "Text_Style_Bold"

  Attribute to enable or disable bold text.
- #define ATTRIBUTE_TEXT_STYLE_ITALIC "Text_Style_Italic"

  Attribute to enable or disable italic text.
- #define ATTRIBUTE_TEXT_FONT "Text_Style_Font"

  Attribute to assign the font to use.
- #define ATTRIBUTE_INPUT_TYPE "InputType"

  Attribute to assign the type of input for the input component.

5.4.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.4.2 LICENSE

5.4.3 DESCRIPTION

Describes common attributes used within the components

5.4.4 Macro Definition Documentation

5.4.4.1 #define ATTRIBUTE_COLLISION_TAG "Collision_Tag"

Collision Tag for representing the given entity.

String identifier for the collision component, to determine how to handle the collision

5.4.4.2 #define ATTRIBUTE_CREATED "bCreated"

Represents whether it has been created.

is false, and set to true when the component's external dependencies have been created and referenced by the components KeyName

5.4.4.3 #define ATTRIBUTE_KEYNAME "KeyName"

Represents a unique keyname.

When referring to components that depend on a given external object, this provides a facility to create and reference that external dependency when it is created.

5.4.4.4 #define ATTRIBUTE_ORIGIN_X "Origin_X"

Represents the center origin for X.

determines the center origin of the sprite, which determines the origin point of positioning, and the origin point of rotation

5.4.4.5 #define ATTRIBUTE_ORIGIN_Y "Origin_Y"

Represents the center origin for X.

determines the center origin of the sprite, which determines the origin point of positioning, and the origin point of rotation

5.4.4.6 #define ATTRIBUTE_POLYGON_POINTS "Attribute_Polygon_Points"

Attribute holding polygon points.

float array used with the polygon type for determining the shape. the number of points(x,y) determines the polygon ex. [x0,y0,x1,y1,x2,y2] –> 3 - point triangle

5.4.4.7 #define ATTRIBUTE_SPRITE_NAME "SpriteName"

Represents the sprite name.

this is the name of the sprite to be used, which is the key string representing a sprite within the sprite manager

5.4.4.8 #define ATTRIBUTE_ZBUFFER "Z-Buffer"

Represents depth.

for drawable components, determines the depth of that object based on this value. ex {0.0 gets drawn before 1.0}

5.5 AE_Events.h File Reference

Macros

- #define INPUT_KEYBOARD_PRESSED "KeyboardPressed"
  Represents the keyboard button pressed event.
- #define INPUT_KEYBOARD_RELEASED "KeyboardReleased"
  Represents the keyboard button released event.
- #define INPUT_MOUSE_MOVE "MouseMove"
  Represents the mouse move event.
- #define INPUT_MOUSE_SCROLL "MouseScroll"
  Represents the mouse scrolling event.
- #define INPUT_MOUSE_PRESSED "MouseButtonPressed"
  Represents the mouse pressed event.
- #define INPUT_MOUSE_RELEASED "MouseButtonReleased"
  Represents the mouse released event.

5.5.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.5.2 LICENSE

5.5.3 DESCRIPTION

The Defintions within this file are used by the InputSystem to determine what type of events to pass to a given InputComponent with one of these parameters representing its input type.

5.6 AE_Utilities.h File Reference

```
#include "AbstractComponent.h"
```

Macros

- #define PI 3.14159265359f

  The math constant Pi.

Functions

- template<class T >
  T Dot (T x0, T y0, T x1, T y1)

  The dot product between two point vectors (x0,y0) and (x1,y1)
- template<class T >
  T Cross (T x0, T y0, T x1, T y1)

  The cross product between two point vectors (x0,y0) and (x1,y1)
- template<class T >
  bool triangleIsCCW (T x0, T y0, T x1, T y1, T x2, T y2)

  Checks if the three points, when traversed performs a CCW movement.
- template<class T >
  T triangle2DArea (T x0, T y0, T x1, T y1, T x2, T y2)

  Returns the area of the triangle (May return negative area)
- bool pointInPolygon (float px0, float py0, const componentFloatArrayType &polygon)

  Checks to see if the point is within the bounds of the polygon.
- bool Test2DLineIntersection (float ax0, float ay0, float ax1, float ay1, float bx0, float by0, float bx1, float by1)

  Checks if the two line segments are intersecting.
- bool checkPolygonLineIntersections (const componentFloatArrayType &polygon1, const componentFloatArrayType &polygon2)

  checks the intersection of the line segments between two polygons

5.6.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.6.2 LICENSE

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see [[http://www.gnu.org/licenses/]].

5.6.3 DESCRIPTION

A set of utilities and globally dependant constants shared

5.6.4 Function Documentation

5.6.4.1 bool checkPolygonLineIntersections (const componentFloatArrayType & *polygon1*, const componentFloatArrayType & *polygon2*)

checks the intersection of the line segments between two polygons

This is a very robust and performance heavy test that checks for the intersection of each line segment of the first polygon to the second polygon.

Is the first polygon Is the second polygon

Returns

Returns whether the two polygons are intersecting, otherwise it returns false.

```
94
                {
95      int numPoints1 = polygon1.size()/2;
96      int numPoints2 = polygon2.size()/2;
97      //next, the edge check is performed between the two polygons
98      for (int i = 0; i < numPoints1; i++) {
99          // treat the rectangle line segment as AB
100          float ax, ay, bx, by;
101          ax = polygon1[i*2];
102          ay = polygon1[i*2+1];
103          if (i != numPoints1 - 1) {
104              bx = polygon1[(i+1)*2];
105              by = polygon1[(i+1)*2+1];
106          }
107          else {
108              bx = polygon1[0];
109              by = polygon1[1];
110          }
111          for (int j = 0; j < numPoints2; j++) {
112              float cx, cy, dx, dy;
113              cx = polygon2[j*2];
114              cy = polygon2[j*2+1];
115              if (j != numPoints2-1) {
116                  dx = polygon2[(j+1)*2];
117                  dy = polygon2[(j+1)*2+1];
118              }
119              else {
120                  dx = polygon2[0];
121                  dy = polygon2[1];
122              }
123
124              //test the line segments, if they intersect
125              // it means they collided
126              if (Test2DLineIntersection(
127                  ax, ay, bx, by,
128                  cx, cy, dx, dy)) {
129                  return true;
130              }
131          }// END for (int j = 0; j < numPoints; j++) {
132      }// END for (int i = 0; i < rectNumPoints; i++) {
133      return false;
134  }
```

5.6.4.2 template<class T > T Cross (T *x0*, T *y0*, T *x1*, T *y1*)

The cross product between two point vectors (x0,y0) and (x1,y1)

Parameters

x0	is the x component of the first vector
y0	is the y component of the first vector
x1	is the x component of the second vector
y1	is the y component of the second vector

Returns

The cross product of the two vectors

```
9                                   {
10      //|x0 y0| = |a b|
11      //|x1 y1|   |c d|
12
13      //ad * bc
14      //x0 * y1 - x1 * y0
15
16      return (x0 * y1) - (y0 * x1);
17  }
```

5.6.4.3 template<class T > T Dot (T *x0*, T *y0*, T *x1*, T *y1*)

The dot product between two point vectors (x0,y0) and (x1,y1)

Parameters

x0	is the x component of the first vector
y0	is the y component of the first vector
x1	is the x component of the second vector
y1	is the y component of the second vector

Returns

The dot product of the two vectors

```
4                               {
5       return (x0*x1 + y0*y1);
6   }
```

5.6.4.4 bool pointInPolygon (float *px0*, float *py0*, const componentFloatArrayType & *polygon*)

Checks to see if the point is within the bounds of the polygon.

NOTE: The polygon is assumed to be a convex polygon. This test is not effective when determining the point within a non-convex polygon, which would require further testing.

Parameters

px0	is the point x component.
py0	is the point y component.
polygon	is a list of points making up the polygon.

Returns

Returns true if the point is within the polygon, otherwise it returns false.

```
37                                                                          {
38     // points represented by [(x0,y0), ... (xn,yn)]
39     int numPoints = polygon.size() / 2;
40
41     int low = 0, high = numPoints;
42     do {
43         int mid = (low + high) / 2;
44         bool bIsCCW = triangleIsCCW<float>(
45             polygon[0], polygon[1],
46             polygon[mid*2], polygon[mid*2+1],
47             px0, py0);
48         if (bIsCCW) {
49             low = mid;
50         }
51         else {
52             high = mid;
53         }
54     } while (low + 1 < high);
55
56     if (low == 0 || high == numPoints) return false;
57
58     return triangleIsCCW<float>(
59         polygon[low*2], polygon[low*2+1],
60         polygon[high*2], polygon[high*2+1],
61         px0, py0);
62 } //END bool polygonPointsInPolyg...
```

5.6.4.5 bool Test2DLineIntersection (float *ax0,* float *ay0,* float *ax1,* float *ay1,* float *bx0,* float *by0,* float *bx1,* float *by1*)

Checks if the two line segments are intersecting.

The check forms two line segments between the given points, and determines whether they are intersecting. This test can be performed on non-convex shapes, but is very inefficient.

Parameters

ax0	is the first x component for the first line segment
ay0	is the first y component for the first line segment
ax1	is the second x component for the first line segment
ay1	is the second y component for the first line segment
bx0	is the first x component for the second line segment
by0	is the first y component for the second line segment
bx1	is the second x component for the second line segment
by1	is the second y component for the second line segment

Returns

Returns true if the two line segments intersect, otherwise it returns false.

```
66                                                  {
67
68     //test if AB and CD overlap
69     float area1 = triangle2DArea<float>(
70         ax, ay,
71         bx, by,
72         dx, dy);
73     float area2 = triangle2DArea<float>(
74         ax, ay,
75         bx, by,
76         cx, cy);
77
78     //both triangles should have opposite windings, check if the
79     // final sign is negative.
80     if (area1 * area2 < 0.0f) {
81         float area3 = triangle2DArea(
82             cx, cy,
83             dx, dy,
84             ax, ay);
85
```

```
86          float area4 = area3 + area2 - area1;
87          if (area3 * area4 < 0.0f) {
88              return true;
89          }// END if (area3 * area4 < 0.0f) {
90      }// END if (area1 * area2 < 0.0f) {
91      return false;
92  }
```

5.6.4.6 template<class T > T triangle2DArea (T *x0*, T *y0*, T *x1*, T *y1*, T *x2*, T *y2*)

Returns the area of the triangle (May return negative area)

The three points form a triangle, and this function returns the area of that triangle. If the triangle was formed with a CCW winding, the area returned will be positive, otherwise the area returned will be negative.

Parameters

x0	is the x component of the first vector
y0	is the y component of the first vector
x1	is the x component of the second vector
y1	is the y component of the second vector
x2	is the x component of the third vector
y2	is the y component of the third vector

Returns

The area of the triangle (+ if CCW, - if CW)

```
32                  {
33      return ((x0 - x2) * (y1 - y2)) - ((y0 - y2) * (x1 - x2));
34  }
```

5.6.4.7 template<class T > bool triangleIsCCW (T *x0*, T *y0*, T *x1*, T *y1*, T *x2*, T *y2*)

Checks if the three points, when traversed performs a CCW movement.

This check is used to determine the winding for the positioning of points on the screen. This is useful in determining the winding, which decides on which way the face of the triangle is facing. Counter-clockwise points represents a face pointing out of the screen, with a clockwise motion representing the face pointing into the screen.

Parameters

x0	is the x component of the first vector
y0	is the y component of the first vector
x1	is the x component of the second vector
y1	is the y component of the second vector
x2	is the x component of the third vector
y2	is the y component of the third vector

Returns

Returns true if the triangle is formed CCW, otherwise returns false.

```
23                  {
24      auto cross = -(Cross<T>(x1-x0,y1-y0,x2-x0,y2-y0));
25      return cross > 0;
26  }
```

5.7 ArmyEngine.h File Reference

```
#include <memory>
#include <functional>
#include "SFML\Window.hpp"
#include "SFML\Graphics.hpp"
#include "Managers.h"
#include "Systems.h"
#include "Entities.h"
#include "Components.h"
#include "AE_Attributes.h"
#include "AE_Events.h"
#include "AE_Utilities.h"
#include "ComponentFactory.h"
#include "EntityFactory.h"
```

Classes

- class ArmyEngine

  Army engine singleton facade. Frontend to the engine.

Macros

- #define WINDOW_WIDTH 800

  Sets the window width.
- #define WINDOW_HEIGHT 600

  Sets the window height.
- #define WINDOW_TITLE "ArmyEngine Game"

  Sets the default window title.

Typedefs

- typedef std::function< int(int)> functionRegisterTemplate

  Function Type for registering global events.

Enumerations

- enum EnumEventType {
  EVENT_CLOSED, **EVENT_RESIZED**, **EVENT_LOSTFOCUS**, **EVENT_GAINEDFOCUS**,
  EVENT_TEXTENTERED, **EVENT_MOUSE_ENTER**, **EVENT_MOUSE_EXIT** }

  Used for the registered global events.

5.7.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.7.2 LICENSE

5.7.3 DESCRIPTION

Main class which should be used as an interface to access the functionality of the armyengine pieces

The only other classes considered for modification are the Managers

5.8 CallbackManager.h File Reference

```
#include <memory>
#include <functional>
#include <map>
#include <string>
```

Classes

- class CallbackManager

  The callback manager is used to store and retrieve callbacks.
- class CallbackFunctionWrapper

  Wrapper to store varying functions as one function type.

Typedefs

- typedef std::function< int(void)> functionBaseTemplate

  Template for a function callback with no parameters.
- typedef std::function< int(int)> functionTemplate

  Template for a function callback with one parameter.
- typedef std::function< int(int, int)> functionEventTemplate

  Template for a function callback with two parameters.
- typedef std::map< std::string, functionEventTemplate > callbackMapType

  Type used to store the list of callbacks.

5.8.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.8.2 LICENSE

5.8.3 DESCRIPTION

The callback manager contains lists of functions which are used as callbacks to components stored within the entities. It includes one parameter, which is the entities current ID the function must also return '0' in order to determine whether it was successful

Given the design of the engine, the callback has full access to all of the managers, and is able to identify the entity it is contained in by the given ID, which makes it very flexible.

Function Wrapper is used to wrap functions which contain only one parameter It wraps it into a function which can take two parameters and discards the second parameter.

5.9 CollisionComponent.h File Reference

```
#include <string>
#include "AbstractComponent.h"
```

Classes

- class CollisionComponent

  The collision component is used to assign a collision bound to an entity.

5.9.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.9.2 LICENSE

5.9.3 DESCRIPTION

The collision component is used to provide an entity with a collision bound, which other collision bounds react to by calling a registered callback. The callbacks are situated within the CollisionManager.

5.10 CollisionManager.h File Reference

```
#include <memory>
#include <functional>
#include <map>
#include <string>
#include <tuple>
#include <list>
```

Classes

- class CollisionManager

  Collision manager used to perform callbacks on collisions.

Typedefs

- typedef std::tuple< int, std::string, int, std::string, bool > collisionParamTuple

  Tuple for storing collision information as a parameter.

- typedef std::tuple< int, std::string, int, std::string > registeredCollisionTuple

  Tuple for storing collision information as a callback.

- typedef std::pair< std::string, std::string > collisionTagTuple

  Pair Tuple that holds two collision tags.

- typedef std::function< int(collisionParamTuple)> functionCollisionTemplate

  Function Type to assign to the callback.

- typedef std::map < collisionTagTuple, functionCollisionTemplate > collisionCallbackTableType

  Type to hold the callbacks.

- typedef std::list < registeredCollisionTuple > registeredCollisionListType

  Type for holding registered collisions.

5.10.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.10.2 LICENSE

5.10.3 DESCRIPTION

Collision manager is used to manage what happens when two entities collide with eachother. Callbacks are added to the collision manager to handle situations where given collision types collide. Depending on the type of collision, the resulting collision may be handled differently.

For example Player tagged "Player" collides with a wall "Physical", which will invoke a collision callback to prevent the character from moving in that direction. It must also be noted that a second callback where the wall collides with the player will also be invoked.

Information for the callback includes a tuple, which includes 5 values. These 5 values should be enough to perform all the functions desired tuple<entity id 1, collisioncomponent name 1, entity id 2, collisioncomponent name 2, bRegistered>

the callback is chosen based on the pair<collisiontag1, collisiontag2> which needs to be registered within the manager.

5.10.4 Typedef Documentation

5.10.4.1 typedef std::tuple<int, std::string, int, std::string, bool> collisionParamTuple

Tuple for storing collision information as a parameter.

tuple for storing

first collided entity id and collisioncomponent name

second collided entity id and collisioncomponent name

last value is a boolean representing if it has stopped colliding.

5.10.4.2 typedef std::pair<std::string, std::string> collisionTagTuple

Pair Tuple that holds two collision tags.

pair tuple holding first collision tag attribute string second collision tag attribute string

5.10.4.3 typedef std::tuple<int, std::string, int, std::string> registeredCollisionTuple

Tuple for storing collision information as a callback.

tuple for storing

first collided entity id and collisioncomponent name

second collided entity id and collisioncomponent name

5.11 CollisionSystem.h File Reference

```
#include <memory>
#include <tuple>
#include "EntityManager.h"
#include "CollisionManager.h"
#include "AbstractSystem.h"
```

Classes

- class CollisionSystem

Macros

- #define **SYNTH_CIRCLE_POINT_NUM** 20

5.11.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.11.2 LICENSE

5.11.3 DESCRIPTION

Contains the collision system used to determine whether two entities with collision components are colliding. If they are colliding, it calls the necessary collision callbacks.

5.12 ComponentFactory.h File Reference

```
#include <memory>
#include "Components.h"
```

Classes

- class ComponentFactory

5.12.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.12.2 LICENSE

5.12.3 DESCRIPTION

Is a factory for creating components.

5.13 EntityFactory.h File Reference

```
#include <memory>
#include "Entities.h"
```

Classes

- class EntityFactory

  Factory for creating entities.

5.13.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.13.2 LICENSE

5.13.3 DESCRIPTION

Is a factory for creating entities. The factory makes sure that each entity is assigned a unique ID.

5.14 EntityManager.h File Reference

```
#include <vector>
#include <list>
#include <memory>
#include <mutex>
#include "AbstractEntity.h"
```

Classes

- class EntityManager

  The entity manager used to store and manager entities.

Typedefs

- typedef std::vector
  < std::shared_ptr< entityType > > entityVectorType

  Type used to store the vector list of entities.
- typedef std::list
  < std::shared_ptr< entityType > > entityListType

  Type used to store temporary lists of entities when retrieving.

5.14.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.14.2 LICENSE

5.14.3 DESCRIPTION

The entity manager is used to store and manage all of the entities that are currently within the game.

5.15 EventManager.h File Reference

```
#include <memory>
#include <list>
#include <vector>
#include "SFML\Graphics.hpp"
```

Classes

- class EventManager

  Used to Poll for events and store them for later retrieval each frame.

Typedefs

- typedef sf::Event eventType

  Type used to events.

- typedef std::vector
  < std::shared_ptr< eventType > > eventListType

  Type used for holding the events within the event managerx.

5.15.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.15.2 LICENSE

Copyright (C) 2013 Benjamin Zaporzan

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see [[http://www.gnu.org/licenses/]].

5.15.3 DESCRIPTION

Used to manage the events within the game. All events are polled through an external source, and passed into the event manager to be retrieved. Any systems that can make use of the events will then retrieve the events and pass these events onto the components that make use of them.

5.16 EventSystem.h File Reference

```
#include <memory>
#include <tuple>
#include <vector>
#include "AbstractSystem.h"
#include "SFML\Window.hpp"
#include "EventManager.h"
#include "CallbackManager.h"
```

Classes

- class EventSystem

  Used to handle global events that affect the entire application.

Macros

- #define REGISTERED_EVENT_CLOSED "REGISTER_EVENT_CLOSED"

  Describes the occurence of closing the application.
- #define REGISTERED_EVENT_RESIZED "REGISTER_EVENT_RESIZED"

  Describes the occurence of resizing the application window.
- #define REGISTERED_EVENT_LOSTFOCUS "REGISTER_EVENT_LOSTFOCUS"

  Describes the occurence of losing focus to the application.
- #define REGISTERED_EVENT_GAINEDFOCUS "REGISTER_EVENT_GAINEDFOCUS"

  Describes the occurence of gaining focus to the application.
- #define REGISTERED_EVENT_TEXTENTERED "REGISTER_EVENT_TEXTENTERED"

  Describes the occurence of text being entered.
- #define REGISTERED_EVENT_MOUSE_ENTER "REGISTER_EVENT_MOUSE_ENTER"

  Describes the occurence of the mouse entering the application window.
- #define REGISTERED_EVENT_MOUSE_EXIT "REGISTER_EVENT_MOUSE_EXIT"

  Describes the occurence of the mouse leaving the application window.

5.16.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.16.2 LICENSE

5.16.3 DESCRIPTION

Contains the event system, which is a system for processing global events that may be consistent over the entire execution.

This is useful for assigning callbacks for things like pressing the close button, or resizing the window.

5.17 PositionComponent.h File Reference

```
#include <string>
#include "AbstractComponent.h"
```

Classes

- class PositionComponent

  Used to describe the position of an entity.

5.17.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.17.2 LICENSE

5.17.3 DESCRIPTION

This file contains the position component, which is used to describe the position of an entity on the screen.

5.18 ShapeComponent.h File Reference

```
#include <string>
#include "AbstractComponent.h"
```

Classes

- class ShapeComponent

  Is used to express the entity as a shape on the screen.

5.18.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.18.2 LICENSE

5.18.3 DESCRIPTION

Shape Component is a lot like sprite component, but has several predefined shapes which don't require a sprite in order to express them visually on the screen.

5.19 ShapeManager.h File Reference

```
#include <memory>
#include <map>
#include <string>
#include "SFML\Graphics.hpp"
```

Classes

- class ShapeManager

  Used to add / store, and then retrieve shapes for shape components.

5.19.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.19.2 LICENSE

You should have received a copy of the GNU General Public License along with this program. If not, see [[http://www.gnu.org/licenses/]].

5.19.3 DESCRIPTION

The shape manager is used to add / store, retrieve shapes that are being used to represent certain entities on the screen.

5.20 SpriteManager.h File Reference

```
#include <memory>
#include <map>
#include <string>
#include "SFML\Graphics.hpp"
```

Classes

- class SpriteManager

  Used to manage the sprites for sprite components.

Macros

- #define DEFAULT_SPRITE "default"

  the default sprite to use when no sprite has been provided
- #define DEFAULT_SPRITE_PATH "../images/default.png"

  path to the default sprite

Typedefs

- typedef sf::Sprite spriteType

  Type to abstract the sprite.
- typedef std::map< std::string, std::shared_ptr< spriteType > > spriteListType

  Type that holds the sprites within the sprite manager.

5.20.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.20.2 LICENSE

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see [[http://www.gnu.org/licenses/]].

5.20.3 DESCRIPTION

Contains the sprite manager, which is a singleton class used to store / add, retrieve shapes that are assigned to ShapeComponents.

5.21 StateComponent.h File Reference

```
#include <string>
#include "AbstractComponent.h"
```

Classes

- class StateComponent

  Used to store entity state (Not currently used)

Macros

- #define COMPONENT_FAMILY_STATE "State"

  The statically assigned family name.

5.21.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.21.2 LICENSE

5.21.3 DESCRIPTION

I haven't found any use for this component, but it is supposed to store state on the given entity. When the state is enabled, the assigned callback is called each frame, but can be limited to the first time it is enabled by setting the appropriate attributes.

5.22 TextComponent.h File Reference

```
#include <string>
#include "AbstractComponent.h"
```

Classes

- class TextComponent

5.22.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.22.2 LICENSE

Copyright (C) 2013 Benjamin Zaporzan

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see [[http://www.gnu.org/licenses/]].

5.22.3 DESCRIPTION

Holds the text component which is used to represent text on the screen.

5.23 TextManager.h File Reference

```
#include <memory>
#include <map>
#include <string>
#include "SFML\Graphics.hpp"
```

Classes

- class TextManager

  Used to store / add, and retrieve text for use with the text component.

Macros

- #define DEFAULT_FONT "Inconsolata"

  The default font to be used if no other font is considered.
- #define DEFAULT_FONT_PATH "../fonts/Inconsolata.otf"

The path to the default font.

5.23.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.23.2 LICENSE

5.23.3 DESCRIPTION

Holds the text manager, which is used to store / add text which is used by the TextComponent.

5.24 TextureManager.h File Reference

```
#include <memory>
#include <map>
#include <string>
#include "SFML\Graphics.hpp"
```

Classes

- class TextureManager

  Used to store / add and retrieve textures for the sprites.

Typedefs

- typedef sf::Texture textureType

  Type used for textures.
- typedef std::map< std::string, std::shared_ptr< textureType > > textureMapType

  Type used to store the textures.

5.24.1 Detailed Description

Author

Benjamin Zaporzan benzaporzan@gmail.com

5.24.2 LICENSE

5.24.3 DESCRIPTION

Holds the textures that are being used by the sprites stored within the sprite manager.

Index