Code Design and Data Structures Project Design Document

By Leonard Andrew Spencer, 2017

Contents

[Introduction 3](#_Toc486163368)

[Design Overview 4](#_Toc486163369)

[Game Design 5](#_Toc486163370)

[Introduction 5](#_Toc486163371)

[Controls and Gameplay 5](#_Toc486163372)

[Level Design 5](#_Toc486163373)

[Container Classes 6](#_Toc486163374)

[Introduction 6](#_Toc486163375)

[Array 6](#_Toc486163376)

[Stack 6](#_Toc486163377)

[List 7](#_Toc486163378)

[Queue 7](#_Toc486163379)

[Deque 7](#_Toc486163380)

[Map 7](#_Toc486163381)

[State Machine 9](#_Toc486163382)

[Entities 10](#_Toc486163383)

[Observer System 11](#_Toc486163384)

[Resource Management 12](#_Toc486163385)

[User Interface 13](#_Toc486163386)

# Introduction

This project is intended to demonstrate my ability to create state machines and data structures, use design patterns, handle errors, and use unit testing.

This project is maintained using Git, and a repository can be found at <https://github.com/LASpencer/aie-DataStructuresProject>

# Design Overview

# Game Design

## Introduction

This project runs a short 2D platformer game. The player controls a character who can walk on other objects, jump over or onto obstacles, and will fall when walking off an edge. The player wins by reaching a door at the end of the level.

## Controls and Gameplay

The player uses the WASD keys to control the character

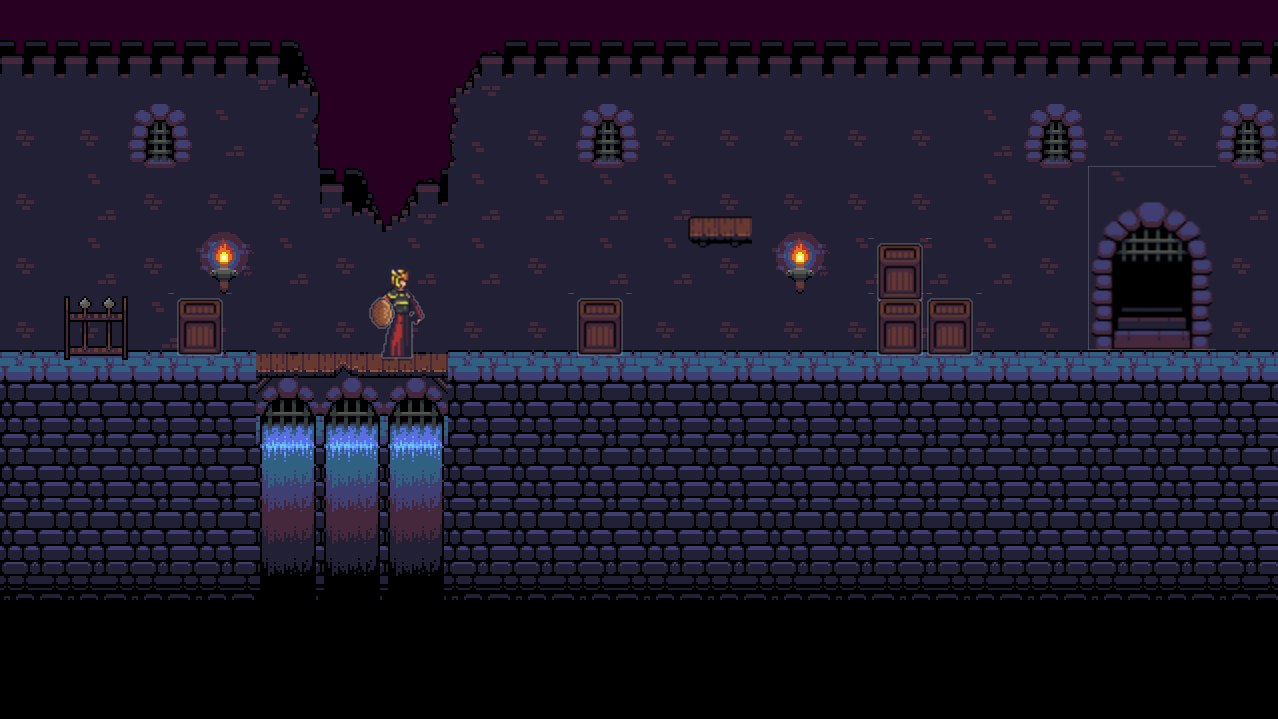
|  |  |
| --- | --- |
| **Key** | **Action** |
| A/D | Move left/right |
| W | Jump |
| S | Crouch |
| ESCAPE | Pause/Unpause |

The character can walk left and right, and move left and right while in the air, but cannot move while crouching. They are able to jump from a standing, crouching, or walking position, but cannot “double jump” while in the air.   
If the character walks off an edge, or is otherwise not supported by some object, they will fall until they land on an object which supports them.

Walking into the door at the end of the level triggers the win screen.

## Level Design

The level was designed to demonstrate basic platforming features. The player needs to reach the door (4) at the right, but the pile of blocks (3) is too high to jump over. So the player has to jump onto the block ahead (1), jump from it onto the platform(2) and then jump onto the pile of blocks(3) to get past them and reach the door(4).



4

3

2

1

Player

Figure Level design

# Container Classes

## Introduction

One of this project’s requirements was that custom container classes, rather than those from the Standard Template Library, were used. Dynamic array (Array), double linked list (List) and search tree map (Map) containers were created to meet this requirement. Stack, Queue, and Deque containers were also created, and implemented in terms of these structures.

Further information on the container classes can be found at <https://github.com/LASpencer/aie-DataStructuresProject/wiki/Containers>

## Array

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Array>

Array is a templated dynamic array, similar to the STL Vector container. Elements can be inserted at or erased from any position, and additional elements can be added to the Array. However, erasing or inserting elements becomes more expensive the more elements there are following this position, as they all have to be moved.  
The Array allocates memory for storing its elements as a contiguous block. By default, it allocates memory for 8 elements. If more elements are inserted than the array has capacity to store, it allocates twice as much memory as it previously had, and copies its elements to the new block. The new block is twice the size of the old one so that the number of allocations required for an array of size n is log n, and the number of copies that need to be made is n log n. If the number of elements required can be estimated in advance, the reserve method can be called to allocate enough memory for those elements beforehand. Also, the shrink\_to\_fit method allows the Array to free unused memory.

The Array class is able to use iterators for some of its functions. As the Array stores its elements as a contiguous block, the pointers to those elements can be used as iterators for the array. This is simple to implement, but does mean that resizing the Array invalidates all existing iterators. More importantly, the Array has templated functions which can take any iterator dereferencing to the type of element contained. These use SFINAE to check the traits of the iterator used. Also, when inserting from a range, the iterator traits are checked to determine if the multipass guarantee holds (that is, if incrementing the iterator does not invalidate its copies). If so, the distance between the first and last iterators is checked, so only one set of copies and resizes needs to take place.

### Stack

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Stack>

The Stack class is a container implemented in terms of an Array. A Stack contains an Array, and each of its methods calls the appropriate methods of the Array. These methods only allow the top element to be added, removed, or accessed.   
There is also a peek method, allowing lower elements in the stack to be accessed as constant references. This allows elements within the stack to be checked easily, while still ensuring it acts as a stack in only allowing changes to the top element.

## List

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/List>

List is a templated double linked list, similar to the STL List container. As a double linked list, elements can be inserted to and erased from any position in constant time. However, accessing an element requires traversing the list from one of the ends to reach it.

The List is implemented using ListNode objects, which contain a value and pointers to the previous and next ListNodes. The List class holds pointers to the first and last ListNodes in the list. These ListNodes cannot be accessed directly, but only through the ListIter class used as an iterator for the List.   
ListIter has references to the List it iterates over, and a ListNode in that list (or nullptr if it’s the end iterator). It dereferences to the value contained in its ListNode. Incrementing and decrementing changes the ListNode to the one pointed to by its current ListNode. Since it contains a reference to the List, the end iterator can be decremented by getting the last ListNode in that list. This could have been done using empty nodes, however several List methods take a ListIter as an argument, and referencing the List in the ListIter allows these methods to check they are only passed ListIters iterating over that List.

### Queue

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Queue>

Queue is a container implemented using a List. It contains a List, and rather than providing that List’s iterators to allow traversal through the list, it only provides access to the value in the front element. Elements can be pushed to the back of the Queue, and popped from the front.

### Deque

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Deque>

Deque is a double ended queue. It is a subclass of Queue, adding methods to enqueue elements at the front (push\_front) and dequeue them from the back (pop\_back).

## Map

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Map>

Map is an associative container, similar to the STL Map container. It is implemented with a Red-Black self balancing tree. As a binary search tree, order by the element’s key, key-value pairs can be found in logarithmic time.

The search tree is made of TreeNode objects. These contain a key and value, as well as references to their parent and child nodes. A TreeNode has two children: left, which has a lower key, and right, which has a greater key. It also has a flag to determine if it should be considered red or black, for balancing the tree.   
The TreeNode class has several methods for traversing the tree. As well as getting its parent and children, it also get its sibling (returning the child of its parent that isn’t itself), traverse up the tree to the root, and find the node with the previous or next key.

TreeNodes have a recursive copySubtree method, for deep copying a TreeNode. This returns a pointer to a copy of the TreeNode it was called on, with that copy having copies of the original’s descendants.

They also have recursive methods for testing whether their tree is a valid red-black tree.

A TreeNode is responsible for is descendants. When destroyed, it will destroy all its descendants and remove its parent’s reference to itself.

The Map contains a reference to the root node of its search tree, and is responsible for it. By destroying this root node, the rest of the tree will be recursively destroyed.

The Map is also responsible for ensuring the tree remains balanced. On inserting a new key-value pair, the new TreeNode containing it is marked as red. If its parent is also red, this indicates that the tree is unbalanced, and recolourings and rotations are performed to resolve this imbalance. The algorithm for doing so was based on the example given here: <http://www.geeksforgeeks.org/red-black-tree-set-2-insert/>

On erasing a node, if it has more than one child, the node’s successor changes places with it. Importantly, these nodes and those referencing them have their references changed, but the nodes keep the same memory location. This ensures that any iterator referencing the successor remains valid. Then, if the node to erase is black and has a black child (or no children, as nullptrs are considered black), a “double-black” occurs, unbalancing the tree. This is resolved by rotating and recolouring until this extra blackness can be moved to a sibling or falls off the top of the tree. The algorithm for resolving the imbalance was based on the example given here: <http://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/>

Maps have an iterator, MapIter. This is incremented and decremented using the referenced node’s getSuccessor and getPredecessor methods, which have logarithmic time. However, this does mean that MapIter is only invalidated by erasing the node they reference. An iterator using depth-first or breadth-first traversal, rather than in-order, could be incremented in constant time. However, these would be invalidated more easily. Some implementations could even be invalidated by incrementing a copy of the iterator (that is, they would be mutable iterators, rather than forward iterators).

# State Machine

## Introduction

One requirement of this project was that game state would be managed using a state machine. This was accomplished using a stack state machine, called GameStateMachine. This state machine contains GameState objects in a Map, and a Stack of keys associate with them. Any state with a key in the stack is updated, with the one on the top being the current state, which controls whether any transitions should occur.

## Implementation

GameState is an abstract base class for the various states used in the state machine. It derives from State and StackState classes, from which it inherits onEnter, onExit, onFocus and onLoseFocus methods. These are called during state transitions, or when a state is pushed on top of that state.

When a state transition needs to occur, the current state sets its shouldTransition, shouldPop, or shouldPush flag to true, and sets its m\_target member to the key to the state which should be transitioned to, or pushed onto it. This can happen in the state’s update loop, or on being notified of an event (See the Observer System section).

GameStateMachine contains a map of registered GameStates (as shared pointers) associated with integer keys. Every update, each state with its key in the stack is updated. Then, the current state’s transition flags are checked, and if one is true the appropriate transition is made. During these transitions, the onExit, onEnter, onFocus and onLoseFocus are called on the states involved.

## Behaviour

## C:\Users\User\AppData\Local\Microsoft\Windows\INetCache\Content.Word\GameState Diagram.jpg

Figure State Diagram for GameStateMachine

### SplashScreenState

This is the first state entered on starting the program. On entry, it begins playing music and loads the splash screen texture. During its updates, it has the ResourceManager load resources which will be needed by later states. After all resources are loaded, if the player presses any key or the splash screen has been displayed for 5 seconds, it transitions to MainMenuState.  
Unlike other states, exiting SplashScreenState does not stop the music, as the same track plays over the MainMenuState

### MainMenuState

This state can be entered from SplashScreenState or WinState. On entry, it begins playing the main menu music (if it’s not already playing) and adds itself as an observer to its buttons (if not already observing them). During updates, an animation of a castle plays. If it receives a click event from either button, it will transition to another state: BattleState for the play button, and FinalState for the exit button. Exiting this state stops the music playing.

### BattleState

This state can be entered from the MainMenuState. On entry, it creates the entities used in the game, starts observing the door entity’s collider, and plays an ambient noises track. During updates, the game is played. If the escape button is pressed, the PauseState is pushed onto the stack. This loss of focus stops the game being updated, and pauses the audio track. These resume on regaining focus.

When this state is notified of a collision between the player and the door, it will transition to the WinState.

On exiting this state (either directly, or from the PauseState causing a transition to the final state), the music is stopped and all entities are cleared from the GameProjectApp’s entity list.

### PauseState

This state can be entered by being pushed onto BattleState. On entry, it begins observing its quit button. The state greys out the screen, and displays a quit button. If the pause button is pressed, this state is popped. If it receives a click event from the quit button, it will transition to FinalState (this also causes any states below it on the stack to be exited).

### WinState

This state can be entered from the BattleState. On entry, it begins observing its buttons and plays victory music. If it receives a click event from the buttons, it will transition to the appropriate state: MainMenuState for the menu button, and FinalState for the quit button. On exit, the music is stopped

### FinalState

This state can be entered from MainMenuState, PauseState, or WinState. It will call the GameProjectApp object’s quit method, shutting down the application.

# Entities

### Introduction

Everything in the game is an Entity object. An Entity is defined by where it is (its position), what it is (its tags), and what it does (its components). This is an example of the Component design pattern. To create entities of a particular type, an EntityFactory class is used, which adds the appropriate components and tags to an Entity object, based on the arguments in its createEntity method. This is an example of the Factory design pattern.

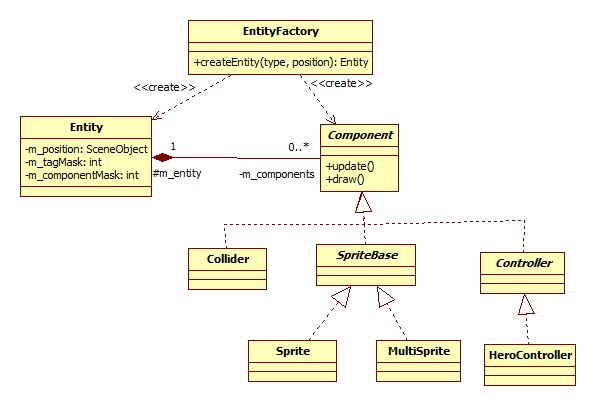


Figure Class diagram showing relationship between Entity, Component, and EntityFactory classes

### Entity Class

The Entity class has an Array containing its components, bitsets to check which components and tags it has, and a SceneObject defining its position.   
During BattleState’s update, the componentMask bitset of each entity is checked to determine which ones have a particular component, and that component is updated for each entity.

### Components

All components are derived from the abstract Component class. There are three component identifiers, which specify a family of Component subclasses that have some particular responsibility. An Entity can have only one Component with a particular identifier.

#### Sprite

The sprite identifier is used by the abstract SpriteBase class, and its subclasses Sprite and MultiSprite. These components draw a sprite to the screen, at the entity’s position. Sprite draws from a single texture, while MultiSprite has an Array of textures, which are drawn over each other.

#### Collider

The collider identifier is used by the Collider class. This component is used for collision detection. It contains an array of Box objects, which have positions for their corners (relative to the entity’s position) and a BoxType specifying what collision with that box means. BattleState tests collision between each pair of Collider components, and passes an Array of any collisions to the static resolveCollisions method, which has both Colliders involved notify their observers.

#### Controller

The controller component is used by the abstract Controller class. This class is for managing and controlling the behaviour of an entity and its other components. The only concrete Controller implemented is HeroController. HeroController sets the sprite’s UVRect and collider’s hitboxes based on the hero’s state.

### EntityFactory

The EntityFactory class is used to create entities. It has a createEntity method, which takes a value specifying the entity to create, and the position to create it at. Based on the entity requested, one of its private methods for creating a specific entity is called.

### Planned Improvements

There are many improvements to this system which could not be implemented due to time constraints. First, it could be expanded to include more things. Rather than having a Button and TextBar class, these objects could be implemented as entities. A component that can display text would be required, and the Collider component would need to also produce event when moused over or clicked on.  
Second, the collision resolution, falling, and landing behaviour in HeroController could be moved out into a Physics component. Attaching this component to an Entity would let it have a velocity set, make it stop, slow down, or bounce on collisions with other entities, and make it fall if not supported. Rather than directly moving the hero entity, HeroController would set the Physics component’s velocity. State transitions for falling and landing could be done through observing the Physics component for events.  
Third, EntityFactory could be able to create different versions of entity types. This could be done through arguments in the createEntity function, or by having methods which could set variables for particular types, such as letting the hero use different textures for their equipment. Another way would be to use prototypes, and have the factory contain a map of prototypes which can be cloned by passing their key to the createEntity function.  
Finally, enemies could be added to the game, which would have their own Controller classes for the AI code controlling the enemy.

# Observer System

The Observer design pattern is used to allow objects to control another object without creating a dependency between them. It involves creating a Subject class which is contains a list of Observers which it can call a notify method on. Based on the arguments of the notify method, the Observer can respond to the Subject’s message, without the Subject needing to know what that response should be, if any.

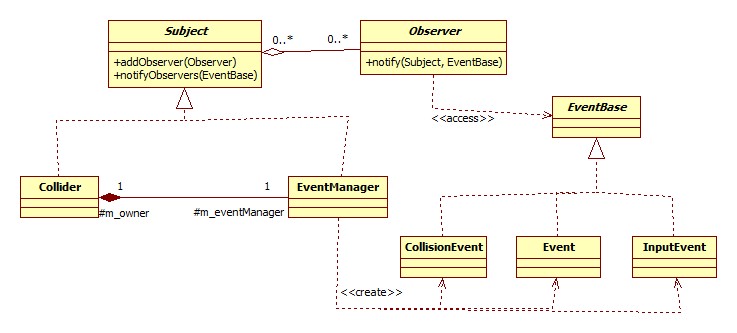


Figure Class Diagram showing relationship between Subject, Observer, and Event classes

### Subject

Subject is a virtual base class for subjects. It provides an interface for adding and removing observers from its list, and notifying all observers that an event occurred.

To allow code reuse, a concrete EventManager class is used by other concrete Subjects to implement this interface. EventManager has a reference to its owner. When notifyObservers is called, it passes a reference to that owner as an argument of the notify function, rather than a reference to itself.

### Observer

Observer is a virtual base class for observers. It provides an interface for notifying the observer of an event. Concrete observers check the Subject and EventBase objects passed as arguments, and based on this may perform some action. For instance, several states transition to a new state on being notified of a button click event.  
As Subjects hold events as weak\_ptrs, Observer inherits from std::enable\_shared\_from\_this to allow it to create shared\_ptrs to itself. So, Observers should be created using std::make\_shared.

### Event

Event objects are created by a Subject and passed by reference to the Observers. They exist in stack memory, and so are destroyed after the notifyObservers function ends.  
Events contain an EventID, describing what event occurred. There are also other Event classes, InputEvent and CollisionEvent, which have more fields to provide information to Observers.

### Planned Improvements

One improvement that could be made is giving Events more methods for testing if their fields match some value. For instance, many observers need to test the BoxTypes and Entity TagMask from a CollisionEvent. A method which takes two BoxTypes and an integer bitmask, and returns whether its boxes and entity match those, would be more convenient than writing the boolean expression to test these every time.

# Testing

This project uses the Catch testing framework for its unit tests.