Nexus - A turn-based battle game incorporating AI

Final Report (Interim)

CS3822 - BSc Final Year Project

Saif Saleemi

Supervised By : Nery Riquelme Granada

Department of Computer Science Royal Holloway, University of London

Table of Contents

1 Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 7108+

Student Name: Saif Saleemi

Date of Submission: 6/12/2022

Signature: SS

2 Abstract

With the role of AI in the gaming industry expected to grow even further in the future I aim to research the fundamentals behind implementing AI in a game and explore game development using the Unity game engine.

By the end of this project I hope to achieve two things; successfully develop and implement an AI that will become more challenging to the player overtime and produce a 3D turn based game using the Unity game engine.

The game itself, Nexus, will revolve around a collection of fighters that the player will be able to pick from as they progress through the game fighting against an AI adversary that will get harder each level. The turn-based action will be similar to games like Pokemon where the player controls a team of characters each containing unique abilities and attributes.

1.2: Reports:

Project reports have been made on key concepts that must be understood in order to progress. Shown below are the concepts that have been researched during this term:

- State-Driven Agent Design
- Goal-Driven Agent Behavior
- Sequencing Patterns
- Behavioral Patterns
- Decoupling Patterns
- Optimization Patterns

The reports for the project above can be found below

1.3: Programs

The Full build for the early deliverable can be found in the Nexus_Early_Deliverable_Build folder.

Proof of concept programs are contained in the Assets folder under Nexus_Early_Deliverable in the directory

3. Report on Behavioral Patterns

This Report will cover Behavioral Patterns that are useful in Game Design

1. Subclass Sandbox

2. Type Object

Subclass Sandbox

Primary Objective:

Define behavior in a subclass using a set of operations provided by its base class

Motivation:

Define a sandbox method (an abstract protected method) that subclasses must implement. Given that, to implement a behavior:

1. Create a new class that inherits its base class

2. Override activate(), the sandbox method

3. Implement the body of that by calling the protected methods that the new class provides

- This fixes redundant code and constrains coupling to one place.

How it works:

A base class defines an abstract sandbox method and several provided operations. Marking them protected makes it clear that they are for use by derived classes. Each derived sandbox subclass implements the sandbox method using provided operations.

When to use it:

- You have a base class with a number of derived classes.

- The base class is able to provide all of the operations that a derived class may need to perform.

- There is behavioral overlap in the subclasses and you want to make it easier to share code between them.

- You want to minimize coupling between those derived classes and the rest of the program.

Design Decisions:

What operations should be provided?

If the provided operations are only used by one or a few subclasses you are adding complexity to the base class which affects everyone, but only a couple of classes benefit.

If the implementation of a provided operation only forwards a call to some outside system, then it isn't adding much value may just be simpler to call the outside method directly

Should methods be provided directly, or through objects that contain them?

The main problem with this pattern is that you can end up with a painfully large number of methods crammed into your base class. You can mitigate that by moving some of those methods over to other classes. The provided operations in the base class then return just one of those objects. This reduces the number of methods in the base class and makes the code easier to maintain. It also lowers coupling between the base class and other systems.

How does the base class get the state that it needs?

Do Two-state initialization:

The constructor will take no parameters and just create an object. Call a separate method defined directly on the base class to pass in the rest of the data that it needs The issue with this is that you need to make sure you always call init(), if you ever forget code will fail. This can be fixed by encapsulating the entire process into a single function that will create an object and initialize it with data

Summary:

This behavioral pattern is a good way to remove redundant code especially in a game design scenario when you are dealing with a lot of entities.

In relation to Nexus this pattern can prove to be useful as we are dealing with a lot of moves (Objects) that will derive from a base class (Moves) which can provide the operations the specific move needs.

<u>Type Object</u>

Primary Objective:

Allow the flexible creation of new classes by creating a single class, each instance of which represents a different type of object.

Motivation:

We want to create objects that have a similar behavior type but varying personal data. The basic OOP approach seems reasonable, making a base class and multiple classes inheriting it but as more and more subclasses are introduced and as we want to fine tune previous subclasses it gets too time consuming. We want to change data without having to recompile the whole game every time.

How it works:

Define a type object and a typed object class. Each type object instance represents a different logical type. Each typed object stores a reference to the type object that describes its type.

When to use it:

1. You don't know what types you will need up front

2. You want to be able to modify or add new types without having to recompile or change code

Problems

Type objects have to be tracked manually, we are responsible for managing not only the classes in memory but also their types. have to make sure all of the type objects are instantiated and kept in memory as long as our classes need them. Whenever we create a new object, we need to ensure that it's correctly instantiated with a reference to a valid type.

Harder to define behavior for each type because we replace an overridden method with a member variable, makes it easy to use type objects to define type specific data but hard to define type specific behavior.

Design Decisions:

Is the type object encapsulated or exposed?

If Encapsulated :

The typed object can selectively override behavior from the type object. Have to write forwarding methods for everything the type object exposes. If our type object class has a large number of methods, the object class will have its own methods for each of the ones we want to be publicly visible.

If Exposed:

Outside code can interact with type objects without having an instance of the typed class.

How are typed objects created?

1. Construct the object and pass in its type object.

2. Call a 'constructor' function on the type object

Summary:

The Type Object pattern is extremely useful in game design as it can accommodate additions fairly easily down the line, for example it would prove to be useful if I wanted to add more attacks or fighters to my game down the line.

However it comes at the cost of needing more management. It could have some use in the development of Nexus however it will not be implemented if not needed as it is a fairly complex pattern to implement and maintain in runtime.

4. Report on Decoupling Patterns

This Report will cover Decoupling Patterns that are useful in Game Design

1. Component

2. Event Queue

Component

Primary Objective:

Allow a single entity to span multiple domains without coupling domains to each other

Motivation:

Different domains should be kept isolated from each other, we want to avoid AI, physics, rendering, sound and other domains to know about each other otherwise any programmer wanting to make a change to the code would need to know something about all intertwined domains to make sure they don't break anything.

We can do this by abstracting domains into separate classes, for example moving all of the code for handling user input into a separate 'inputcomponent' class. Then the target class will own an instance of this Component, this can be repeated for all domains the target class touches.

Our component classes are now decoupled, and if any domains need to interact with each other they can be handled on a case by case basis. This also turns our component classes into reusable packages which can be used for other classes. This method of reusing code is now preferred compared to inheritance which is often too cumbersome for simple code reuse.

How it works:

A single entity spans multiple domains. To keep the domains isolated, the code for each is placed in its own component class. The entity is reduced to a simple container of components.

When to use it:

1. You have a class that touches multiple domains which you want to keep decoupled from each other

2. A class is getting massive and hard to work with

3. You want to be able to define a variety of objects that share different capabilities, but using inheritance doesn't let you pick the parts you want to reuse precisely enough

Design Decisions:

How does the object get its components?

If the object creates its own components:

1. Ensures the object always has the components it needs.

2. Harder to reconfigure the objects

If outside code provides components:

1. Object becomes more flexible

2. Object can be decoupled from the concrete component types.

How do components communicate with each other?

By modifying the container object's state:

1. Keeps the components decoupled

2. It requires any information that components need to share to get pushed up into the container object

3. Makes communication implicit and dependent on the order that components are processed. Need to be careful with the order components are laid out in the update method

By referring directly to each other:

1. simple and fast, communication is a direct method call from one object to another

2. The two components become tightly-coupled, although its not as bad as having all code in a single class.

By sending messages:

1. Most complex option, can create a telegram system into our container object that lets components broadcast information t o each other. Can define a base component interface that all components will implement which has a single receive method that component classes implement in order to listen to an incoming message. Then can add a send method in our containerobject. Now if a component has access to its container, it can send messages to the container, which will rebroadcast the message to all contained components.

2. Ensures the components are decoupled from each other, the only coupling being the message value.

3. Container object is simple, all it does is blindly pass messages along.

Summary

The unity framework core GameObject class is designed entirely around the component pattern. This pattern is a good option to consider during production of Nexus, it can be used when you decouple user input code from a main class, for example the user input with their fighter.

Event Queue

Primary Objective:

Decouple when a message or event is sent from when it is processed.

Motivation:

Game code is likely complex enough as it is. The last thing we want to do is stuff a bunch of checks for triggering something like in-game tutorials in there. Instead you could have an event queue where any game system can send to it and receive events from it so the actual game and tutorial is decoupled from each other and a certain event from the event queue triggers the tutorial.

How it works:

A queue stores a series of notifications or requests in FIFO order. sending a notification enqueues the request and returns. The request processor then processes items from the queue at a later time. Requests can be handled directly or routed to interested parties. This decouples the sender from the receiver both statically and in time.

When to use it:

Is good to use when you want to decouple something in time (during runtime)

Code Structure

Ring buffer queue implementation (circular queue) with the head of the Queue is where Requests are read from. The head of the queue is the oldest pending request. Tail is the slot in the array where the next enqueued request will be written

Design Decisions:

What goes in the queue?

If you queue events:

An event is something that has already happened

1. Likely to allow multiple listeners

2. Scope of the queue tends to be broader (more globally visible)

If you queue messages:

message describes an action that we want to happen in the future

1. More likely to have a single listener

Who can read from the queue?

A Single-cast Queue:

1. Queue becomes an implementation details of the reader
2. Queue is more encapsulated
3. Don't have to worry about contention between listeners

A Broadcast Queue:

1. Events can be disregarded (if there are 0 listeners)
2. May need to filter events to reduce amount of event handlers to invoke

A Work Queue:

1. You have to schedule, since an item only goes to one listener queue needs logic to figure out the best one to choose.

Who can write to the Queue?

One Writer:

1. Implicitly know where the event is coming from
2. Usually allow multiple readers otherwise it will just feel like a basic queue

Multiple Writers:

1. Have to be more careful of cycles because a feedback loop can occur
2. Likely want to reference to the sender in the event itself, since there are multiple writers, the reader needs to know who sent the event.

Summary:

Event Queue pattern is a good way to decouple when we want decoupling to happen. This can be useful in the creation of Nexus, for example for adding a tutorial where the player is given a brief introduction on how to play the game during the game. This is common in a lot of AAA games where user input triggers 2 events at once; the tutorial and the base game event.

5. Report on Goal-Driven Agent Behavior

This Report will cover Goal-Driven Agent Behavior and will answer the following questions:

1. How does it work?

2. How can it be implemented?

We will then look at the advantages of implementing Goal-Driven Agent Behavior

How does it work?

In this pattern, an agent's behavior is defined as a collection of hierarchical goals which are either atomic or composite.

Atomic goals are goals that define a single task, behavior or action whereas composite goals are composed of several subgoals which in turn may be atomic or composite creating a nested hierarchy.

Each update the agent examines the game state and selects, from a set of predefined high-level goals, the one it believes will most likely enable it to satisfy its strongest desire. It will then attempt to follow this goal by decomposing it into any constituent subgoals, satisfying each one in turn doing this until the goal is either satisfied or has failed, or until the game state necessitates a change of strategy.

How can it be implemented?

Firstly we must make use of the composite design pattern which works by defining an abstract base class that represents both composite and atomic objects which enables agents to manipulate goals identically, no matter how simple or complex they are.

Goal objects are similar to the state class, having a method for handling messages and the Activate, Process and Terminate methods are similar to the Enter, Execute, and Exit methods of state.

The Activate method initializes logic and represents the planning phase of the goal which can be called any number of times to replan if the situation demands it.

The Process method is executed each update step, returning an enumerated value indicating the status of the goal which can be one of four values:

Inactive - goal is waiting to be activated

Active - the goal has been activated and will be processed each update step

Completed - The goal has completed and will be removed on next update

Failed - Goal has failed and will either replan or be removed on next update.

The Terminate method undertakes necessary tidying up before a goal is terminated and is called just before a goal is destroyed.

To tidy up implementation, a lot of the logic can be abstracted out into a Goal_Composite class, which all concrete composite goals can inherit from. All composite goals will call a function 'ProcessSubgoals' each update step to process their subgoals, ensuring that completed and failed goals are removed from the list before processing the next subgoal in line and returning its status.

Goal Arbitration:

Using the highest-level goal Goal_Think, which each agent owns a persistent instance of, an agent can arbitrate between available strategies (goals), choosing the most appropriate to be pursued.

Every ‘think’ update each of these strategies is evaluated and given a score representing the desirability of pursuing it. The strategy with the highest score is assigned to be the one the agent will attempt to satisfy.

To calculate a score, each Goal_Think aggregates several Goal_Evaluator instances, one for each strategy. These objects have methods for calculating the desirability of the strategy they represent and for adding that goal to Goal_Think's subgoal list. Each CalculateDesirability method is a hand-crafted algorithm that returns a value indicating desirability of a bot pursuing that strategy. Algorithms can be hard to create so making helper functions that map feature specific information from the game to a numerical value in the range 0 to 1 which are then utilized in the formulation of the desirability algorithms.

Create your own formula for specific strategies, for example, a goal GetHealth can have a formula of desirability -> (D = k * 1 - Health / DistToHealth) where k is a constant used to tweak the result. This relationship makes sense because the farther you have to go to retrieve an item the less you desire it, whereas the lower your health level, the greater your desire. Look for similar relationships in strategies to find a good formula to calculate desirability. Algorithms can be improved in this case, as right now it is a linear function which isn't realistic. To create a nonlinear function divided by the square or even cube of DistToHealth.

Goal_Think

Goal_Think iterates through each evaluator each think update and selects the highest to be the strategy a bot will pursue. Goal_Think is the arbiter of strategy goals. You can even switch in and out entire sets of strategy goals to provide an agent with a whole new suite of behaviors to select from. This is used to good effect in AAA games like Far Cry.

Benefits of using Goal-based Arbitration Design

Goal Arbitration is an algorithmic process defined by a handful of numbers. It is driven by data instead of logic like in FSM. This is very beneficial as all you have to do to tweak behavior is change numbers.

Another advantage of hierarchical goal-based arbitration design is that extra features are provided with little additional effort from the programmer.

Personalities:

We can create agents with different personality traits by multiplying desirability scores with a constant that biases it in the required direction (Aggressive agent that prioritizes attacking over defending).

To facilitate this, the Goal_Evaluator base class contains a member variable Char_Bias, which is assigned a value by the client in the constructor, this is then used in the CalculateDesirability method to adjust the score. To make personalities persist between games we can create a separate script file for each bot containing the biases.

State Memory:

The stack-like nature of composite goals automatically endows agents with a memory, enabling them to temporarily change behavior by pushing a new goal (or goals) onto the front of the current goal's subgoal list.

As soon as the new goal is satisfied it will be popped from the list and the agent will resume whatever it was doing previously. Even if the agent strays off path the built-in logic for detecting failure and replanning allows the design to move backward up through the hierarchy until a parent is found that is capable of replanning the goal and putting the agent back on track.

Command Queueing:

Players can order agents to do multiple things one after the other, for example they can order an NPC to Patrol between two points. This is heavily used in RTS games where players can think ahead and send NPCs to do multiple goals.

Summary:

Goal-Driven agent design is a flexible and powerful architecture allowing agent behavior to be modeled as a set of high-level strategies, each of which is comprised of a nested hierarchy of composite and atomic goals.

All agents will have a goal Goal_Think which is their highest level goal that will help them arbitrate between other strategies/goals using desirability scores and bias.

Although it shares many similarities, this type of architecture is far more sophisticated than a state-based design providing a lot of advantages at the cost of being more complex to code and implement compared to its State-design counterpart.

How can it be applied to Nexus?

This design can have many benefits in the production of Nexus. The AI the player will play against can have a series of goals it wants to satisfy such as keep the fighter alive or kill an enemy fighter. It can also possess different personality traits by adding bias to desirability scores. For example it could be an aggressive AI that focuses on using damage based attacks instead of defense providing more diversity for the player to fight against.

We can also change the difficulty of the AI by creating entire sets of strategy goals to provide the agent with completely new behavior. For example, an advanced AI may have goals and subgoals that decide to use an attack move that is strong against the enemy fighter type and will make more intuitive decisions such as using a heal item when low on health or defense moves. A novice AI will just have atomic goals such as kill fighters which can be done by using high damage abilities disregarding whether they are effective or not against that specific fighter.

6. Report on Optimisation Patterns

This Report will cover Optimisation Patterns that are useful in Game Design

1. Dirty Flag

2. Object Pool

3. Spatial Partition

Dirty Flag

Primary Objective:

Avoid unnecessary work by deferring it until the result is needed

Motivation:

For example, we want to collapse modifications to multiple local transforms along an object's parent chain into a single recalculation on the object. Also avoid recalculation on objects that didn't move. And if an object gets removed before its rendered, it doesn't have to calculate its world transform at all. Essentially we can speed up computation by not performing operations on objects we know will not change.

How it works:

A set of primary data changes over time. A set of derived data is determined from this using some expensive process. A “dirty” flag tracks when the derived data is out of sync with the primary data. It is set when the primary data changes. If the flag is set when the derived data is needed, then it is reprocessed and the flag is cleared. Otherwise, the previous cached derived data is used.

When to use it:

This pattern solves a pretty specific problem. Should only reach for it when you have a performance problem big enough to justify the added code complexity.

Design Decisions:

When is the dirty flag cleaned?

When the result is needed:

1. It avoids doing calculations entirely if the result is never used.

2. If the calculation is time-consuming, it can cause a noticeable pause.

At well-defined checkpoints:

1. Doing the work doesn't impact user experience, can give something to distract the player while the game is busy processing.

2. Lose control over when the work happens.

In the Background:

1. Can tune how often the work if performed

2. Can do more redundant framework

3. Need support for doing work asynchronously.

How fine-grained is your dirty tracking?

If it's more fine-grained:

1. Only process data that actually changed

If its more coarse-grained:

1. End up processing unchanged data

2. Less memory used for storing dirty flags

3. Less time is spent on fixed overhead.

Summary:

Although fairly complex to implement, this pattern can provide a very good way to speed up the efficiency of the game. This type of pattern works fairly well with FPS games where physics plays a part.

Object Pool

Primary Objective:

Improve performance and memory use by reusing objects from a fixed pool instead of allocating and freeing them individually.

Motivation:

Programming games is similar to programming embedded systems, memory is scarce. We want to make sure that creating and destroying objects doesn't cause memory fragmentation. An object pool can solve this problem. To the memory manager, we're just allocating one big hunk of memory up front and not freeing it while the game is playing. To the users of the pool, we can freely allocate and deallocate objects to our heart's content.

How it works:

Define a pool class that maintains a collection of reusable objects. Each object supports an "in use" query to tell if it is currently "alive". When the pool is initialized, it creates the entire collection of objects up front (usually in a single contiguous allocation) and initializes them all to the "not in use" state.

When you want a new object, ask the pool for one. It finds an available object, initializes it to "in use", and returns it. When the object is no longer needed, it is set back to the "not in use" state. This way, objects can be freely created and destroyed without needing to allocate memory or other resources.

When to use it:

1.You need to frequently create and destroy objects

2.Objects are similar in size

3.Allocating objects on the heap is slow or could lead to memory fragmentation

4.Each object encapsulates a resource such as a database or network connection that is expensive to acquire and could be reused.

Design Decisions:

Are objects coupled to the pool?

If objects are coupled to the pool:

1.Implementation is simpler

2.You can ensure that the objects can only be created by the pool.

If objects are not coupled to the pool:

1.Objects of any type can be pooled. can create a generic reusable pool class

2.The “in use” state must be tracked outside the objects

What is responsible for initializing the reused objects?

If the pool reinitializes internally:

1.The pool can completely encapsulate its objects.

2.The pool is tied to how objects are initialized.

If outside code initializes the object:

1.The pool’s interface can be simpler.

2.Outside code may need to handle the failure to create a new object.

Summary:

This pattern can improve performance of a game greatly and can work fairly well with Nexus. For example when we have a large amount of objects to deal with we can look to implement this pattern if it makes things more efficient. This will likely be the case as we will have a lot of objects in the game including the environment, the enemy fighters and the player's fighters.

Spatial Partition

Primary Objective:

Efficiently locate objects by storing them in a data structure organized by their positions.

Motivation:

If we store our objects in a data structure organized by their locations, we can find them much more quickly. This pattern is about applying that idea to spaces that have more than one dimension.

Spatial partitions exist to knock an O(n) or O(n^2) operation down to something more manageable. The more objects you have, the more valuable that becomes. Conversely, if your n is small enough, it may not be worth the bother.

How it works:

For a set of objects, each has a position in space. Store them in a spatial data structure that organizes the objects by their positions. This data structure lets you efficiently query for objects at or near a location. When an object's position changes, update the spatial data structure so that it can continue to find the object.

When to use it:

You have a set of objects that each have some kind of position and you are doing enough queries to find objects by location that your performance is suffering.

Design Decisions:

Is the partition hierarchical or flat?

If it's a flat partition:

1.It's simpler (flat data structure)

2.Memory usage is constant

3.Can be faster to update when objects change their positions

If it's hierarchical:

1.It handles empty space more efficiently.

2.It handles densely populated areas more efficiently

Summary:

This pattern trades memory for speed. If you're shorter on memory than you are on clock cycles, that may be a losing proposition. We can make use of this pattern if we ever get into a situation where we need to keep track of the location of objects instead of using a naive approach like a detection system.

7. Report on Sequencing Patterns

This Report will cover Sequencing Patterns that are useful in Game Design

1. Double Buffer

2. Game Loop

3. Update Method

Double Buffer

Primary Objective:

To cause a series of operations to appear instantaneous or simultaneous

Motivation:

Makes things like rendering seem instantaneous to create the illusion of a coherent game world

How it works:

Two framebuffers, one represents the current frame (the one the video hardware is reading from) and the other is where our rendering code is written to . When rendering is done a switch() function is used to swap the framebuffers making the other framebuffer available to use.

When to use it:

1. We have some state that is being modified incrementally

2. That same state may be accessed in the middle of modification

3. We want to prevent the code that's accessing the state from seeing the work in progress

4. We want to be able to read the state and we don't want to have to wait while its being written

Summary

The core problem double buffering solves is the state being accessed while it is being modified. This has 2 causes :

1. The state is directly accessed from code on another thread or interrupt.

2. When the code doing the modification is accessing the same state that it's modifying. This can occur in a lot of situations such as in AI where entities are interacting with each other

Game Loop

Primary Objective:

Decouple the progression of game time from user input and processor speed

Motivation:

Provide immediate feedback instead of having to wait for results creating an interactive program.

How it works:

The game loop processes user input but doesn't wait for it, the loop always keeps going, and is usually implemented as follows:

```
while(true){
processInput(),
Update(),
Render()}
```

ProcessInput() - handles user input that has happened since the last call

Update() - advances the game simulation one step (runs AI and physics

Render() - Draws the game so the player can see what happened

Overall the Game Loop does two main things:

1. Run the loop shown above

2. Tracks the passage of time to control the rate of gameplay

When to use it:

This pattern is fairly special because it is the core of every game and most game engies such as the one being used to create Nexus (Unity) already have the game loop hardcoded into their engine.

Update Method

Primary Objective:

Simulate a collection of independent objects by telling each object to process one frame of behavior at a time

Motivation:

Each entity in a game should encapsulate its own behavior. This will keep the game loop uncluttered and make it easy to add or remove entities. To do this we add the update() method, the game loop maintains a collection of objects but doesn't know their concrete types. All it knows is that they can be updated separating each object's behavior both from the game loop and from the other objects. Game loop goes through the collection of entities and calls update() on each, giving each a chance to perform one frame worth of behavior. The game loop has a dynamic collection of objects making it easy to add or remove them from collection

How it works:

The game world maintains a collection of objects. Each object implements an update method that stimulates one frame of the object's behavior. Each frame the game updates every object in the collection.

When to use it:

Good to use when a game has a wide range of live entities that the player interacts with, but bad to use when the game is more abstract and the moving pieces are less like living actors and more like pieces on a chessboard.

What to do if the order in which objects are updated is important:

The order in which objects are updated are important. For example, if A comes before B in the list of objects, then when A updates, it will see B's previous state. But when B updates it will see A's new state since A has already been updated.

Updating sequentially each update incrementally changes the world from one valid state to the next with no period of time where things are ambiguous and need to be reconciled. To stop new objects acting during the frame it was spawned can cache the number of objects in the list at the beginning of the update loop and only update that many before stopping: This is done by incrementing an integer that represents the length of the array of objects and cache the length in numObjectsThisTurn at the beginning of the loop so the iteration stops before we get to new objects. **Design Decisions:**

What class to put the update() method in?

1. Entity Class - This is the simplest option and works if you don't have too many kinds of entities, but having to subclass Entity every time you want a new behavior can be painful when you have a large number of different kinds.

2. Component Class - This lets each component update itself independently, lets you decouple parts of a single entity from each other. More on this will be researched in Chapter 5 (Decoupling Patterns)

3. A Delegate Class - Other patterns involve delegating part of a class's behavior to another object. The State pattern does this and the Type Object pattern.

How do you deal with dormant objects?

To deal with dormant objects you can maintain a collection of live objects that need updating, when an object is disabled it's removed from the collection.

The problem with this is that using separate collections for active objects takes extra memory, and there is still the master collection. Works ok when speed is more important than memory. This can be improved further by having the other collection only contain inactive entities instead of all of them, requires collections to be kept in sync.

Overall the more inactive objects you have the more useful it is to have a separate collection that avoids them during game loop

Summary

In this report we have researched 3 useful sequencing patterns used in game design: Double buffer, The Game loop and the Update() method. In relation to Nexus, we can already expect not to be coding the game loop ourselves as it is already hardcoded in the unity game engine but we can apply the knowledge learnt about the double buffer pattern and the update method to aid us in creating a well coded game.

8. Report on State-Driven Agent Design

This Report will cover State-Driven Agent Design and will answer The following questions:

1. Why are Finite State Machines Useful In Game Design?
2. How Can Finite State Machines be Implemented in Game Design?

After These questions have been answered we will delve deeper into the code structure of Finite State Machines in games and apply this knowledge to Nexus.

Why are Finite State Machines (FSMs) Useful in Game Design?

Firstly to answer this question we must know what a Finite State Machine (FSM) is. A FSM is a device which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. Another thing to note is that FSMs can only be in one state at any moment in time. The objective of a FSM is to decompose an object's behavior into easily manageable chunks or 'states'.

With this being said we can now identify how FSMs can be useful in game design. By decomposing object behavior it makes actions easy to code/debug and adds flexibility. For these main reasons FSMs are seen as the general backbone of AI programming.

How Can Finite State Machines be Implemented in Game Design?

The basic approach to implementing a FSM is to use multiple If-statements or a switch statement, this proves to be infeasible for more complicated programs as these programs will consist of far too many states. Therefore this method has a lack of flexibility as adding states down the line will prove to be challenging.

A more effective approach would be to use State Transition Tables (STTs) which can be queried by an agent (object) at regular intervals and react based on the stimulus it receives. Each state can be modeled as a separate object or function existing external to the agent making it clean and flexible. All rules in the table are tested each time interval and if needed the state object is inputted into the agent to update the agent's behavior.

Additional Information

Before any FSM is coded it is imperative that a *BaseGameEntity* class is made. This is a simple class with a private member for storing the ID number which makes every entity/agent unique. All agent classes will derive from this base class.

There are two State types that will be useful in the creation of Nexus:

1. Current State - The state the object is currently in
2. Global State - A state which can be triggered by every other state
3. Previous State - Can save the previous state while changing state

General Code Structure

Each State object will have an Entry method and an Exit method which are called when the object is entering or leaving that state. Create a State base class *State* which all state objects will derive from

Have a *State Machine Class* which keeps the design alot cleaner by encapsulating all the state related data and methods, separating them from agent classes. This allows an agent to own an instance of a state machine and delegate management of current states, global states and previous states

The State Change Process is as follows :

1. Record Previous State
2. Call Exit Method of Current State
3. Change to New State
4. Call Entry Method of New State

One Final thing to note is that we can use multiple FSMs working in parallel, for example one to control a character's movement and one to control the weapon. This Structure is known as Hierarchical State Machine and will prove to be useful during game development

Use of Messaging

A great way to indicate a change of state is for an agent to send/recieve a message. This enhances the illusion of intelligence a great deal. How to do this is shown below:

Telegram Structure

To deal with Message Dispatch and Management a Class *MessageDispatcher* will have a function *DispatchMessage* which creates a message.

Before a message can be dispatched, the *MessageDispatcher* must obtain a pointer to the entity specified by the sender. To do this there needs to be a database of instantiated entities provided for the *MessageDispatcher* to refer to. The *EntityManager* contains a map in which pointers to entities are cross-referenced by their ID.

Message Handling

Steps needed to incorporate Message Handling:

1. Edit *BaseGameEntity* so any subclass can receive messages, done by declaring a pure virtual function *HandleMessage*
2. Edit *State* so *BaseGameEntity* states can choose to accept and handle messages, done by declaring pure virtual function *OnMessage*
3. Edit *StateMachine* so it contains a *HandleMessage* method, When an entity receives a telegram it is sent to the entities current state and then the global state if needed

HandleMessage and *OnMessage* are bools to indicate that the message has been received successfully.

How this all ties in with Game Design and Nexus

Overall we have gained some insight into FSMs and can see how to implement them and the general code structure needed to maintain a FSM. We can now start to model some prototype FSMs for Nexus which will be expanded and improved later down the line with the use of State-Transition Diagrams.

- FSM #1 *Menu* - Can have different states for menu options (Settings, NewGame,etc.)
- FSM #2 *Game* - Can have different states for the game (Start, Finish)
- FSM #3 *Turn* - Can have different states for the turn (PlayerTurn, OpponentTurn)
- FSM #4 *Fighter* - Can have different states for the fighter (Alive, Dead, InUse, WaitingForMove, MakingMove, FinishedMove, etc.)
- FSM #5 *Attack* - Can have different states for the Attack (Usable, NotUsuable, etc.)

Reference:

Chapter 2 of Programming Game AI by Example - Mat Buckland

9. Bibliography

[1] **Nystrom, Robert - Game Programming Patterns**

https://gameprogrammingpatterns.com/ - Provides structural patterns required to create a well coded game.

- Research targets 4 Main chapters and their respective sub chapters :
    1. Sequencing Patterns - (Double Buffer, Game Loop, Update Method)
    2. Behavioral Patterns - (Bytecode, Subclass Sandbox, Type Object)
    3. Decoupling Patterns - (Component, Event Queue, Service Locator)
    4. Optimization Patterns - (Data Locality, Dirty Flag, Object Pool, Spatial Partition)

[2] **Buckland, Mat - Programming Game AI By Example**

http://www.ai-junkie.com/books/toc_pgaibe.html - Provides useful AI design patterns and examples of implementation in games.

- Chapters Researched are shown below:
    1. State-Agent Driven Design
    2. Goal-Driven Agent Behavior (Chapter 9)

[3] **Unity Technologies** - https://unity.com - Provides a vast array of learning resources to help understand Unity and a wide variety of assets to use in game production.