Applying Design Patterns in Game Programming

Junfeng Qu[1], Yinglei Song[2], Yong Wei[3]
[1] Department of Computer Science & Information Technology, Clayton State University, Morrow, GA, 30260
[2] Department of Computer Science, Jiangsu University of Science and Technology, Zhenjiang, China, 212001
[3]Department of Computer Science, North Georgia State University, Dahlonega, GA 30597

***Abstract*—This paper discussed an object-oriented design for general game using C# and XNA using design pattern. We presented application of structural patterns, creational pattern and behavioral pattern to create game sprite, manage game state and game sprites, different collision and rewards among sprites or between sprites and map; we also discussed how to apply design patterns to handle communications between sprites and NPC by using observer pattern and mediator patterns. Although lots of design patterns are discussed, other design patterns might suitable as well because game programming are so complicated to separate each pattern independently.**

Keywords-Game, Programming, Design Patterns, UML, XNA, C#

I. INTRODUCTION

A. Computer Game and Development

The video games industry has undergone a complete transformation in recent years especially after mobile device and casual game have impact people's life greatly.

Computer programmers are writing game in the way, that cow boys are riding on wild west, wild and innovative. However, as the game is getting bigger, more complex, and changing during development. A well designed overall game program design and architecture that modulated and integrated with software development procedure are very important. Also, a well- designed game program should be able to extend easily, portal to other platform easily without deep revision of source code to minimize deliver time is also important.

A large size of program can be developed and organized better with Object-oriented Programming(OOP) because of its significant advances over procedure programming. A series of new techniques and packages have been proposed to handle the complexity and organization problems in game programming, for example, XNA from Microsoft, AndEngine for android mobile game, etc. These components are usually context insensitive and can be used to work on most general game related programming issues and programmers are able to concentrate on the part of the code that often defines the functionalities of game.

B. Design Patterns

Design patterns are proven solutions to well-established software engineering problems. In game programming, programmers are often tend to make sure the correctness of a program by evaluating its behavior of character, and overlooked the design aspect, such as open-close principle, scalability, maintainability, flexibility, extensibility, and robustness to changes, therefore, programmer has to rework or dispose their work complete in order to accommodate changes of algorithm, level, and game mechanics during the game development process.

Due to their well-known importance and usefulness, we proposed some example design pattern solutions to these commonly problems encountered during game development with Microsoft XNA such as handle sprite, communication, control, and collision.

It's not easy to find patterns that can be used as common solutions for common problems in game programming. There are two categories of design patterns in game development. One category of design pattern was introduced by Bjork[1], where a set of design pattern is used for describing(employing a unified vocabulary) the game mechanics(gameplay and game rules) during game development. It focuses on reoccurring interaction schemes relevant to game's story and core mechanics of game. After interviewed with professional game programmers, the authors analyzed the existing games and game mechanics and then proposed those patterns involving game design process. The authors said 'The way to recognize patterns is playing games, thinking games, dreaming games, designing games and reading about games'. For example *Paper-Rock-Scissor* pattern is commonly known in game as triangularity, and this pattern was used in game when there are three discrete states, or options as described in figure 1.

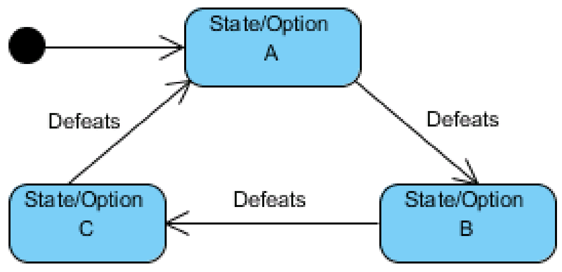

Figure 1. Triangularity design pattern in game

These patterns are not related to the software engineering , architecture or coding. So these are not discussed in this paper. The second category of design pattern in game is use of object-oriented design patterns in programming games, which are discussed and analyzed in the following sections.

One of the unique characteristics of game development and programming is rapidly evolutional modification and goal changing during game design and development, therefore it's very common that game programmers have to dispose their works that they have been working for months and to restart again. Once the game has been completed, it is often transformed into various game platform, such as PC, mobile devices (Android, iOS, Windows 8 etc), game console (PS3, Xbox 360 etc.). Therefore a well-designed game program would spend minimal efforts and changes to migrate. A well designed game programming that offer great flexibility, code reusability, extensibility, and low maintenance costs is highly desired.

Daniel Toll etc.[2] found that it is difficult to perform unit testing in computer game. Computer game often involves poor defined mathematical models, therefore it's difficult to produce expected results of unit under testing. On the other hand, computer game's rules of play needs to validated based on player's inputs, and new functions are unlocked as player makes progress, which in term makes it's difficult to perform testing in the complex interactions of varieties of game objects. For example, as player is making progress in Angry Birds, new challenges features are unlocked to entertain and challenge player, and player is able to perform more options and actions to overcome challenges presents. As these levels, new game items, and new features are added into game, even a small change of codes results a number of test and retest large part of the game. The difficulties of testing of game are also because the tight coupling of modules in game programming.

Most research works focus on teaching design patterns using game programming as examples, and show how effectively there are represented in case studies, such as computer game[3], the Game of Life[4], the Game of Set[5] and [6], which uses a family of games to introduce design patterns. Some researchers[7] had evaluated the usage of design patterns in game programming. It has proven that if design patterns are used properly and in appropriate cases, the programming maintainability, extensibility, flexibility and comprehensibility can be extremely beneficial and improved.

In this paper, we discuss some design patterns in the category of creational patterns, structural patterns and behavioral patterns, such as builder pattern, strategy pattern, mediator pattern, and state pattern, and how these are adapted into game programming info-structure such as C# and XNA.

II. GAME ARCHITECTURE AND LOOP

A. Game Architecture

Most computer games shares a similar architecture in regardless of languages used in game development. Bishop[8] etc. described a general software architecture of game as shown in Figure 2.

The slid-line ovals represent essential component of game architecture and the dashed-line ovals are modules that can be found in more complex games. The Even handler and the input provides player's action to game. The game logic renders game's core mechanics and story if any. The audio and graphics supplies sounds, images, game objects etc. in the game world to the player based on the level data module, where the details about static behaviors are stored. The dynamic module configures the dynamic behavior of game's character and objects. Most official games have all or partial components of above architecture, such as UnReal, Unity 3D, RPG Maker etc.

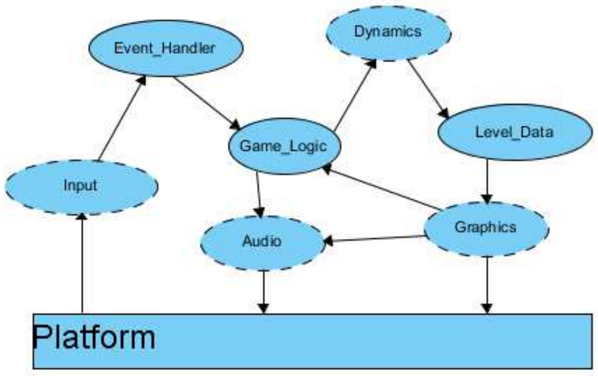

Figure 2. Common Game Architecture[8]

B. Game Loop in Game Programming

In general, most game programming can be viewed as an game loop. The player's inputs are process in each iteration, and the game states and the game world change based on internal game logics until the game is over. Of course the rendering and game logic processing can be coded with event thread, which leads to a simpler code. In small scale or turned-based game with little or no animation, this approach works perfectly. Visual C Sharp XNA provides a game loop that is driven by a control loop that similar to the event-processing loop described above. The game loop uses active rendering as shown in figure 3.

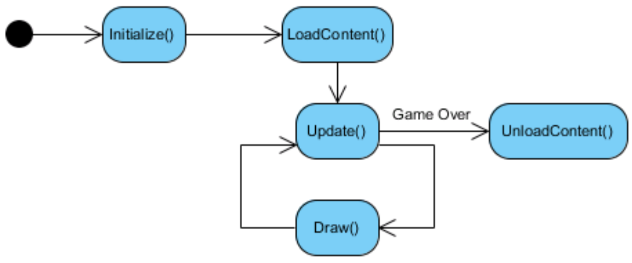

Figure 3. Game Loop Template in XNA

Game *initialization()* include nongraphics initialization. *LoadContent()* include graphics initialization, such as reading game object, sprite, texture etc. After that *Run()* is called to initiate game loop, which includes *Update()* and *Draw()* methods.

Update() method updates game objects, checking for collisions, game AI, game object movement, updating scores, checking for end-game logic etc. *Draw()* method is used to draw game objects on game scene. All logics that effects the gameplay will be done in the loop of *Update()* and *Draw()*. If game ending logic is satisfied, *UnloadContent()* is call to unload resources and memory allocated to game scene. In *Update()* and *Draw()* of the game loop following game related objects are handled:

- Player's inputs: The player's inputs from keyboard, mouse, game console are process and saved into system
- Game internal logic: This is a key component of game. Game rule is implemented in this loop as well. The new game state is decided once upon player's inputs are received and processed based on rule the game designer's plan.
- All game objects in the game scene is update at certain predefined frame-rate based on player's inputs as well.

In this paper, we have proposed a couple of design patterns that we have experienced during game development and design since it's very apparent in game development the common elements and mechanics that the games share are often handled with class abstraction, inheritance, polymorphism in code refactoring.

We use Microsoft XNA as a game development platform and try to integrated creational patterns, structural patterns and behaviors patterns into XNA game loop described above. Design patterns can be applied in design and coding of any game module, what we have illustrated here does not imply that these patterns are more suitable and applicable than other patterns or fields since game programming is so complicated to be included in all scenarios in the discussion, and it also does not mean no other design patterns can be used.

III. Applying Design patterns in Game Programming

A. *Game State Management: State Pattern*

Almost every game starts with a state of an introduction, then move to some kinds of menu such as setting of game or a learning mode, and then player can start play and game enters into playing state. During the playing of the game, the player will be able to jump back to main menu, set parameters, or pause the game until the player is finally defeated and the game moves to a game-over state, the player then may start from main menu again. In general, each state handles different events differently, from, and draw something different on the screen. Each state might handle its own events, update the game world, and draw the next frame on the screen differently from other game states. Figure 4 illustrated an example game state change from main entry to Play State, Pause state and End State respected to different button that pressed by the player.

Traditionally, the multiple states of game are handled with a serious of *if..else if..* statement, *switch..case* statement. Every time through the game loop, the game program must check current state of the game and display and draw game objects correspondingly, also, events are handled and checked to see player's input will trigger the change of game state. This programming approach results a highly coupled codes, therefore it's difficult to debug, testing and code maintain.

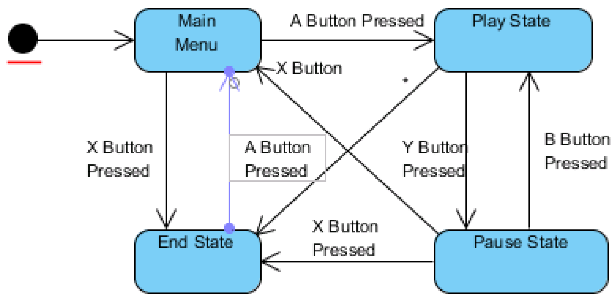

Figure 4. An Example of Game State Changes

State pattern is a natural solution to above problems as illustrated in Figure 5. The state pattern allows an object to alter its behavior when its internal state changes[9].

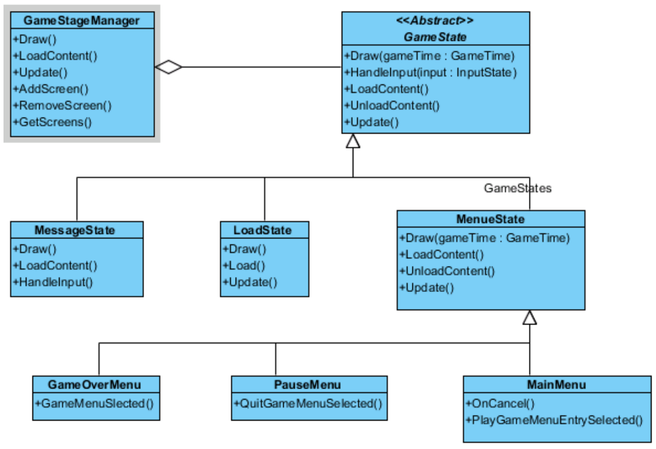

Figure 5. State Pattern for Game State Management

GameStateManager maintains a concrete state at any given time. The abstract *GameState* class encapsulates the behavior associated with a particular state of game. The concrete states of game such as *LoadState, MenuState, Pause, EndState, and Main* implement the behaviors associated with each state in regarding *Draw(), Update()* respectively.

With the use of state pattern, first, we avoided excessively and repetitively using of *switch* .. *case* or *if* .. *else*, therefore the complexity of the programming is reduced, secondly, the application of state pattern explained software engineering principles such as Open-Closed principle and single responsibility principle. Each game state is a subclass, in case more states are required during game development, the programmer simply adds a subclass, e.g. programmer will be able to create a subclass to manipulate background of game. In case the state requirements are changed, the programmer just modifies the corresponding class. Thirdly, the benefit of use state pattern is that the classes are well encapsulated, the change of state is implemented within each class, caller does not to need to know how changes of state and behavior are implemented internally. Lastly, the state objects can be shared if they have no instance variables. State objects protect the context from inconsistent internal states, because state transitions are atomic (the transition between states happen by changing only one variable's value, not several)[9]. Although state pattern brings so many benefits, the complicated game might produce too many subclasses quickly to be out of the control of the programmer and it might be so difficulty to manage these classes.

B. Creation and Behavior of Game Objects: Factory, Command , and State Patterns

In Microsoft XNA game programming, all graphics, sounds, effects, and other items are loaded in XNA thought content pipeline. A sprite in XNA is a flat, preloaded image that is used as part of a computer game, such as players, enemies, and projectiles. To draw a sprite on game world, programmer needs to specify location information that tells XNA where to draw the image as well as where the resource is located in the OS. In XNA, *Texture2D* is one of most commonly used sprite to render images in game world. The Sprite itself lend to object-oriented design: it has states and exhibits behaviors as well.

Sprites have state and they exhibit behaviors. The state of a sprite includes information of location, velocity, size and image. The behavior of sprites usually is based on external or internal game information and modified itself input for player sprites, or gameplay.

The program used nested loop with if or switch statements to explicitly detect the current state and take the appropriate behavior. This procedural approach carries with it all the usual baggage: State-dependent logic is distributed throughout the code and adding new state is error-prone.

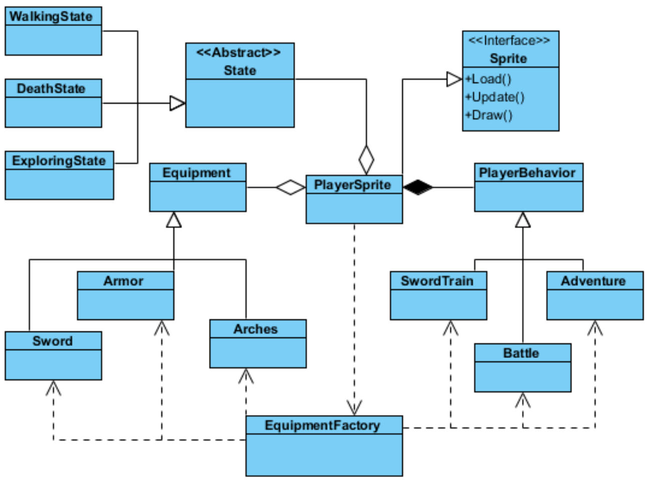

Figure 6. Factory, State and Command Patterns in RPG

In RPG game, a character, player or enemy is often represented by a sprite has to face difference challenges and act correspondingly with different behaviors, for example player may work on training to use sword, complete an mission or submission of a battle, or even adventure to hunt for treasure. Of course it's possible to implement above behaviors within sprite with loop and/or switch, the open-closed principles is not quite followed in above approach. Base on GoF, Command design pattern encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. A command object can have a lifetime independent of the original request, and can specify, queue, and execute request at different times[9]. The command pattern encapsulates the player's behavior as an object to facilitate extends of player's behavior. By specifically creating a behavior class to solve a variety of behaviors a player may have. We can deal with evaluation of game design easily. If a now behavior is needed for game development and story, a new class that inherits from behavior class can be added to implement concrete actions that player need to work on.

In RPG game, player also often equipped with different equipment based on game development and player's progression. It's natural to create and equipment superclass that can be concretely implemented with different equipment such as sword, armor etc. Factory pattern can have an object return an instance from a family of related classes[9]. The player behaves differently based on game development and game progression, for example, the sword and armor are used in training, arches is used during adventure, therefore, an *EquipmentFactory* class is introduced to determine what equipment are required according to different scenario, which is strategy pattern. The strategy patterns defines a family of algorithms, encapsulates each one, and make them interchangeable[9]. By employing these patterns, the program code can be maintained easily and it's more flexible to accommodate changes in game development such as behavior change, equipment adding and removal based on scenarios.

Of course, the state pattern can be deployed as well as illustrated in UML of figure 6, where player may experience walking state, death state, exploring state etc. that can be extended easily after inherits is superclass State.

C. *Game Object Collision and Communication: Visitor, Observer Pattern and Mediator Pattern*

A variety of game objects often collide with each other. Depends on types of game object, it can be collision between a sprite to other sprite, or sprite collides with background map. For example, in games, it's quite common that player's object collides with other different object to receive different credits based on game rule. To entertain the players better, game designers often add a variety of game objects to increase play of fun in game to reward players unexpectedly. The NPC is often introduced as well to work with player or fight against player, either case, the state of player is necessary to broadcast to the teammates or interest game objects based on game mechanics.

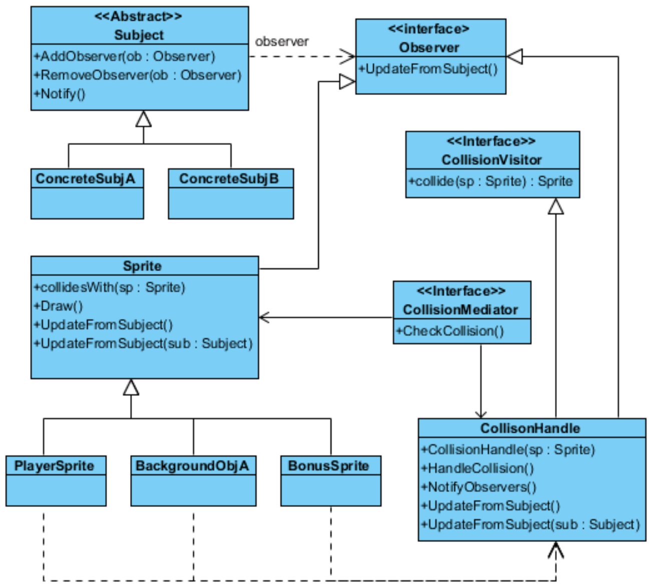

Figure 7. Visitor, Observer and Mediator for Collision and Communication

Visitor design pattern represents an operation to be performed on the elements of an object structure. Visitor lets programmer define a new operation without changing the classes of the elements on which it operates[9]. Visitor pattern is suitable when you want to be able to do a variety of different things to objects that have a stable class structure. Adding a new kind of visitor requires no change to that class structure, which is especially important when the class structure is large. By using of visitor pattern, different collision algorithms can be implemented and different rewarding rules of a variety of objects collision can be implemented while following open-closed principle of software design. The UML illustrated in figure 7 shows that CollisonVisitor interface handles different collision among different sprite in game world.

In RPG game, the character sprite changes states, for example, 'Live' and 'Dead', the domain must notify the graphical user interface to allow it to update itself. Likewise, when the user clicks on, or collides with other objects, the UI must notify the domain so that it can record the appropriate changes to its model.

To communicate among sprites of interests, observer pattern or mediator patter are illustrated in figure 7. Depends on communication is one-to-many or many to many, programmer could choose one or both to pass different subject to interested game elements. According to GoF, The observer pattern is applicable and appropriate in many situations including when (1) The application has two separate aspects that can be varied independently of one another, or (2)the application involves objects that when changed require changing other objects. In observer pattern, a list of watcher(observers) are notified any time the state of the subject changes. The observer pattern defines a one-to-many dependency between objects that when one object changes state, all its dependents are notified and updated automatically. The abstracting coupling between subject and observers make it easier to update notifications to be broadcasted and as a result the subject is not interested in which observers care about the changes, since it is their responsibility to react to it[9]. The observer pattern allows programmer vary subject and observers independently. The subjects can be reused without reusing their observers, and vice versa.

Mediator pattern promotes the many-to-many relationships between interacting peers to "full object status". The Mediator pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently[9]. The communications between objects are encapsulated in mediator, and objects are no longer directly communicating with each other, but rather through the mediator. If mediator pattern is used for communication among game objects, the mediator will be responsible to update game objects. The mediator handles communications between all of these objects to reduce coupling between game objects when the sprite might collide with one another under certain circumstances as illustrated by the UML in figure 7.

IV. Conclusion

In this paper, we have presented the use of a family of design patterns in game development that can be integrated with XNA game development well during game programming. We have covered design patterns that could be used to create sprite, separate behaviors from sprite with strategy and command patterns, separate states from sprite by using state patterns, game state management with state design pattern, communication among sprite with observer or mediator patterns, and collision detection with the visitor pattern. Additionally, the applicability of other design patterns in game development should be also investigated as well.

To evaluate the benefits of object-oriented design patterns in game, we plan to conduct a software quality metrics analysis in terms of size, complexity, coupling and cohesion in near future.

REFERENCES

[1] S. Björk, S. Lundgren, H. Grauers, and S.- Göteborg, "Game Design Patterns," *Lecture Note of the Game Design track of Game Developers Conference*, 2003.

[2] D. Toll and T. Olsson, "Why is Unit-testing in Computer Games Difficult?," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 373–378.

[3] P. V. Gestwicki, "Computer games as motivation for design patterns," *ACM SIGCSE Bulletin*, vol. 39, no. 1, p. 233, Mar. 2007.

[4] M. R. Wick, "Teaching Design Patterns in CS1 : a Closed Laboratory Sequence based on the Game of Life," in *SIGCSE*, 2005, pp. 487–491.

[5] S. Hansen, "The Game of Set – An Ideal Example for Introducing Polymorphism and Design Patterns," in *SIGCSE*, 2004, pp. 110–114.

[6] M. A. Gómez-Martín, G. Jiménez-Díaz, and J. Arroyo, "Teaching design patterns using a family of games," *ACM SIGCSE Bulletin*, vol. 41, no. 3, p. 268, Aug. 2009.

[7] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology*, vol. 49, no. 5, pp. 445–454, May 2007.

[8] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz, "Designing a PC Game Engine," *Computer Graphics inEntertainment*, no. February, pp. 2–9, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995.