Proceedings of The National Conference
On Undergraduate Research (NCUR) 2016
University of North Carolina Asheville
Asheville, North Carolina
April 7-9, 2016

Team Development of Model View Controller Software in the Unity 3D Engine

Nicholas Blum, W. Henry Henderson, Michael Parker, Michael Kuczkuda, James Yount, Sean Stamm, James Morrow

Computer Science
One University Heights
The University of North Carolina at Asheville
Asheville, North Carolina 28804 USA

Faculty Advisor: Dr. Adam Whitley

Abstract

This software development project implements a board game, King of Tokyo, using object oriented (OO) abstractions and MVC (Model View Controller) architecture. Group software development is often lacking in undergraduate computer science curricula, and so this project focuses on teaching software design skills, team skills, and good coding practices. The game is constructed with the C# language in the Unity engine, self-described as "a flexible and powerful development platform for creating multi-platform 3D and 2D games and interactive experiences." The engine allows for quick testing and has many well documented instructional resources and tutorials in C#. It uses a component based methodology, which involves the use of renderers, scripts, and controllers to create game objects. This project combines OO development methodologies with Unity's standard component-based game design. The software's MVC architecture provides a separation between the game state (model) and the graphical representation of the game (view), which allows the team to work on both modules simultaneously and easily make design changes. Unity game objects implement this view by displaying graphics and notifying the controller of user input. OO features of C# such as interfaces facilitate this logical separation between the model and the view. The first prototype is functionally complete, including all major game mechanics. Ideas for future work include incorporating terrain data from Google Maps, adding network multiplayer, AI players, and providing a unified aesthetic for the user experience.

Keywords: Controller, Unity 3D Engine, Model

1. Introduction

KaijuKO is a software implementation of the board game King of Tokyo in the Unity game engine. Developed throughout several semesters since spring 2015 by groups of students, KaijuKO teaches the importance of teamwork as well as good software practices when working in larger, scalable projects. These software practices include adhering to many of the beneficial features of object oriented programming (OOP). In addition, the project codebase is written in model-view-controller (MVC) style. The project also takes advantages of the features of the game engine it is constructed in, Unity, to promote quick prototyping and testing. This introduction will begin by describing the basic rules of the game itself before moving on to a quick overview of MVC, OOP, and the benefits of Unity as a game engine.

King of Tokyo is a king-of-the-hill board-game, where players controlling Godzilla-size monsters (Kaiju) compete for control of the city of Tokyo. To win, a player must either collect a sufficient amount of victory points, or kill all other monsters. During their turn, players roll dice that give them victory points and energon (in-game currency), as well as provide damage points for attacking other players. Players can use energon to purchase cards. Cards have a wide variety of effects ranging from simple damage to other players to generating interest on existing money or

stealing another player's cards. Although the way the game is won is the same, the path to success changes every game as different strategies become viable depending on the cards that are available.

Because the game has so many cards it was important to discover a way to maintain the state of the game while also allowing the project to scale up easily. As more and more cards were added, a well-architectured and designed system would ensure that the new effects would not conflict with the old ones while being easy to implement. In addition, the team for the project consists of individuals with different focuses and specializations such as artists, coders, and modelers. And finally, as the project becomes more and more complex, code conflicts are inevitable, which can slow down development. Once the game's logic was constructed in C# and the project members began to discuss the best way to implement the game in Unity, they chose MVC in large part due to the way that it satisfies these issues.

The software's MVC architecture provides a separation between the game state (model) and the graphical representation of the game (view). A controller script handles communication between the two separate systems. This allows the view, implemented in Unity, and the model, implemented in C#, to work together while being worked on independently. The model could be implemented in many different views, just requiring a new controller to work correctly. Project members can work on the model and view separately, cutting down on code conflicts between teams of programmers and allowing artists to model while programmers code. This is also very helpful because the view in Unity is component-based whilst the model in pure C# is much more object oriented. The controller and MVC enable these two different paradigms to work together reliably.

The Unity game engine is powerful, intuitive, well-supported and easily code-tested [1]. The language chosen for the project, C#, is one of the major languages used for Unity development, and has an extensive library of supporting functions. In addition, Unity has a large, helpful community which can answer questions relevant to game development. Unity's C# programming community is much larger than that for Unity's other scripting languages, Javascript and Unityscript, which is one major reason C# was chosen for the project [2]. C# also has many of the benefits of typed languages, such as easy polymorphism and class hierarchy as well as the benefits of functional languages such as lambda expressions and untyped variables. It strikes a good balance, which is excellent for game development code, which needs to both be easily scalable and easily readable.

In contrast to traditional OO, Unity follows the scheme of a game object as a pure aggregation [3]. The objects or entities in the game world are collections of different components, such as Renderers, Scripts, and Controllers. Combining those components into discrete objects allows the developer to build complex interworking objects out of simple reusable pieces. This is great for game development because it means you can build a single piece of code, for instance a Card script, and attach it to different types of Cards. These cards, although containing the Card script, can behave differently depending on what other components exist on the objects proper.

2. Structure of Project Code

As shown below in figure 1, the code follows a rather concrete hierarchy. Classes can have one or more subclasses that relate to things that class might control. For instance, a deck will contain cards. The project utilizes MVC, so each of the pieces communicates using contracted rules in the form of interfaces. This section will briefly discuss the model, view, and controller before moving to a discussion of several key game systems.

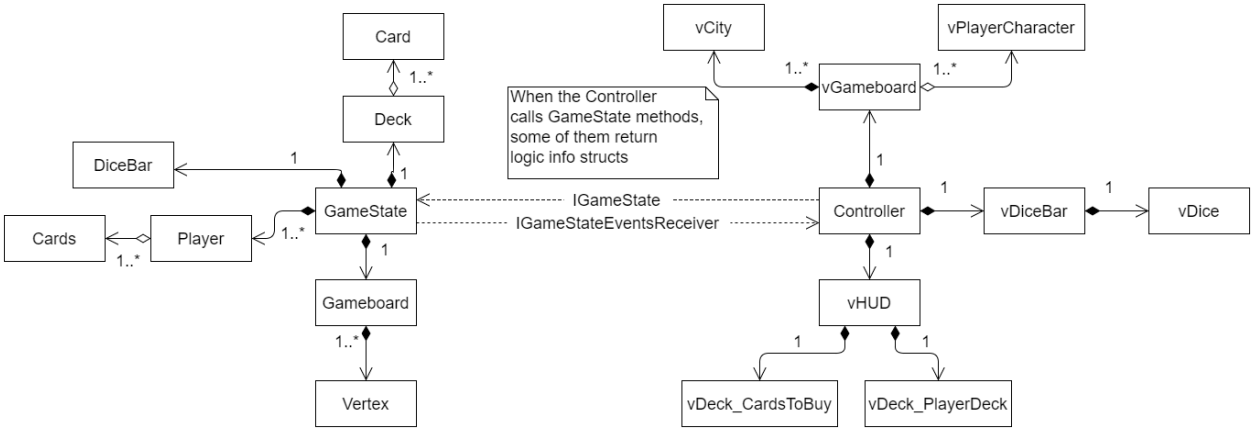

Figure 1. Class diagram of project code

2.1. Model

The model structures classes hierarchically, and is independent of the Unity implementation. The GameState class contains the interface methods as well as inter-communication methods for the logic subclasses (eg, a method to query the game board for the player in Tokyo).

The Player class contains each player's current resources stored in integers: health, energy, and victory points. In addition, each player has a deck of cards and a card event manager to handle any applied effects on that player. Resources are not modified directly. Instead, the class contains functions that are called with an initial amount and check to see if the player has any effects that would modify that amount before performing the modification. For example, if the player has a card that doubles any points received, the ModifyPoints() function will take in the initial points, see that card, and double the points before applying them.

GameBoard keeps track of which players are at what location. Target locations are stored as vertexes. At the moment, the only important location vertex is 0, which is Tokyo proper. In the future, however, this system will allow the developers to expand the game board with other locations that might apply different effects (for example, a rainy location, windy, et cetera). The game board uses an adjacency matrix so that when a player tries to move in or out of a city, the game board can first verify that the desired city vertex is not already occupied.

The DiceBar class updates on a turn-by-turn basis and contains the current state of the active player's dice. It also contains the values of each die, encoded into a character array. Players get a total of at most three rolls per turn. After rolling, a player may choose to keep certain dice and reroll the others. A Boolean array is mapped to the dice array so that when the dice are re-rolled, DiceBar can hold back any dice which the player wants to keep. It can be dynamically resized at any time to handle cards that modify the dice bar.

A card's effects is called through ICardEffect which interfaces the GameState class. This allows the cards to update the model, performing various game-changing tasks such as modifying player health, seizing Tokyo from another player and gaining a second turn. Many of the methods used to create these effects are abstracted through the GameState class to reduce redundancy and facilitate easier card creation through use of existing methods.

The Card class is a polymorphic model implementation through its three subclasses for each card type: passive, instant, and active. These three types of cards inherit their properties from the main Card class as well as instantiate all card objects when the game begins to build a random deck of cards. Instantiation is performed using the reflection features of C#—all classes inheriting from Card that are not abstract are pulled into an array and one of each is stored in the deck when the game begins. This allows programmers to continue to add and develop cards without worrying about needing to add each one to the deck - it is all taken care of automatically.

2.2. Controller

The controller manages communication between the model and the view. As detailed in the figure above, information can flow both ways, and this is facilitated through the use of interfaces. The model can inform the controller of events it should respond to using IGameStateEventReceiver, and the controller can query or update the state of the model through IGameState. The controller implements the singleton design pattern during construction so that any major part of the view can communicate with it without requiring a reference.

The controller plugs into the Unity ecosystem so that it can see both the view and the model. When the game begins, the controller generates the game state using the information gathered in the main menu. The main menu collects the names that the players want, the monsters that they want, and the number of total players. Once the game state logic has completed construction, the controller generates the view by querying it for information, and continues this pattern until the game is completed.

2.3. View

Players interact with the view in order to play the game. As part of the MVC contract, it mirrors the model at all times so that players know what they are looking at is the actual state of the game. To this end, the view contains two systems of scripts. Scripts in the first system mirror the model exactly and are denoted with a v in front of their name. For instance, for every Card there is a vCard. In addition, the second system contains scripts for elements of the user interface that provide additional information, such as a turn order widget or a notification bar.

In order to maintain the view's synchronicity with the model, all arrays are indexed on the model and indexes can only change by querying the model. For instance, the list of a player's vCards must always be in synch with the list of their cards, otherwise, they will try to play one card and end up playing another!

The beauty of Unity's component-based system as well as MVC decoupling means that at any the controller can destroy or empty the view and re-populate it with new or updated information. Therefore instead of writing a function to update every individual element of the UI from the controller, a data structure containing all the necessary information moves from the model to the view. The view will handle populating its subcomponents and objects using that data structure.

All view scripts are Unity monobehaviour, meaning that they live in the game loop. The designers can code animations, graphics, and UI elements so that all the controller has to worry about is calling Player.Move(); the view script handles the subsequent transformations and rotations for moving the player model on-screen. Once again, this allows the project's modelers, animators, and designers to work on making the project look good while the programmers can focus on making the project work well.

3. Subsystems of Code

KaijuKO has several important systems to modify the state of the game. These include an EndTurn system, a CardEvent system, and a system to handle user input. These systems are quite complex and involve multi-threading, complex data structures, and finite state machines.

3.1 CardEvent

Cards are what make each individual game of KaijuKO unique. No two games are the same because the draw pile is randomized every time. Cards are complicated and can make major modifications to the state of the game as well as act differently depending on what other cards the players have. In addition, players can have effects applied that are independent of the cards themselves or that last after the card is used. This necessitated the abstraction of the card effects from the cards proper. Therefore, this system includes several structures: GameEvents, CardEffects, and a CardEffectManager.

A GameEvent occurs at a specific point in the game codebase, such as when the dice are rolled, for any players which might be affected (E.g., All players under attack receive a unique OnHitEvent storing the attacking player and damage being dealt). The class contains information about that event as well as functions for card effects to modify it. For example, when a player is attacked an OnHitEvent is generated that stores the damage that the player will receive. That player might have a card that reduces damage by 1. The effect will reduce the damage stored in the event. In effect, each player's events are stored and modified individually before being applied to the overall game logic itself.

Events are responded to and modified by CardEffects, which are are ongoing effects that are applied to players. All card effects inherit the abstract class CardEffect, and override the base doEffect method with their own functionality. These doEffect methods can contain numerous calls to the ICardEffect interface, which allows the code to do conditional checks based on the state of the game and any other effects currently applied to the player. All effects respond to different events, so the game will only process effects at specific times in the codebase. A CardEffect responding to OnHit will only respond when the player is attacked.

Therefore, each player has a CardEffectManager class that handles any effects applied to that player. The functions in an effect manager are called with an event as a parameter, and handle looping through any CardEffects that player has which respond to that event. One of the most important features of the manager is a hidden stack which allows the model to handle multiple concurrent events. Figure 2 below demonstrates just one possible stack that could occur during normal gameplay.

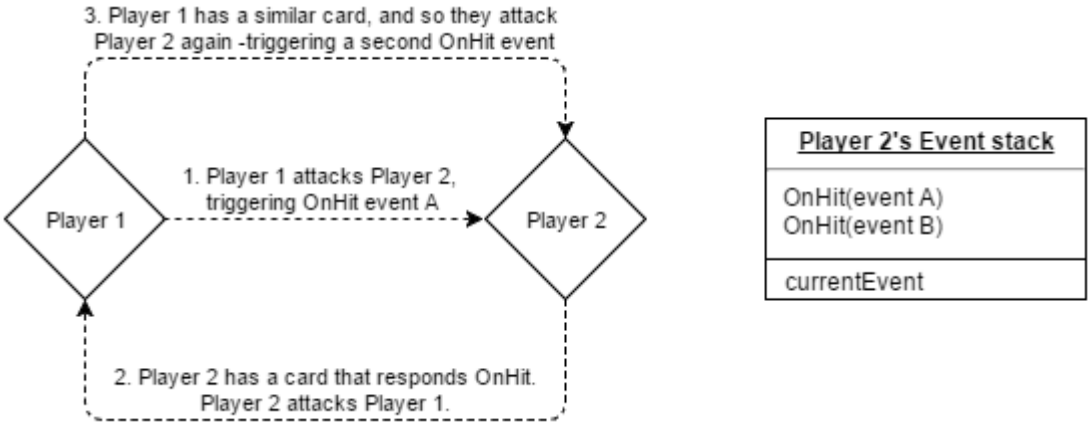

Figure 2. UML Activity diagram detailing a possible event stack during gameplay.

In this attack chain, player 1 attacks 2, generating an OnHit event, A. 2 has a card, "Spiked Shield" that does damage back to player 1. Player 1 also has this card, which means that there is now another attack coming back at player 2, event B. Card effects will need to respond to both of these events in order. In order to do this, the CardEffectManager has a function that adds the event to the stack and then loops through all card effects that respond to OnHit. These card effects will use currentEvent as a pointer to determine which event to respond to / modify. This function will be called once for event A, and then a second time for event B. Event B is pushed onto the stack, and then currentEvent will point to event B. Effects will respond to that, and then at the end of the function, event B is removed. The game then returns to the function handling event A, and currentEvent points to that event once more.

3.2 EndTurn

The end of turn is one of the most complicated pieces of logic in the game. When an attack ends, the game has to handle damage dealt to players from the dice roll. Sometimes damage is not dealt, and sometimes it is dealt but there is no one on the receiving end (for instance, a player attacks Tokyo but no one is there). Besides that, the game checks for player deaths and game over conditions to determine if a new turn should be started, or the game should be concluded. This causes several possible branches which during a previous semester were implemented in the manner of a finite state machine. However, the new card event / effect system implemented this semester required the developers to take a hard look at how events were responded to. Some events only fire in response to user input. In the old system, user input would happen after the game logic checked and dealt damage. As more cards were added that could modify / deflect damage, it quickly became evident that a more robust system was needed. The implementation is detailed in figure 3 below.

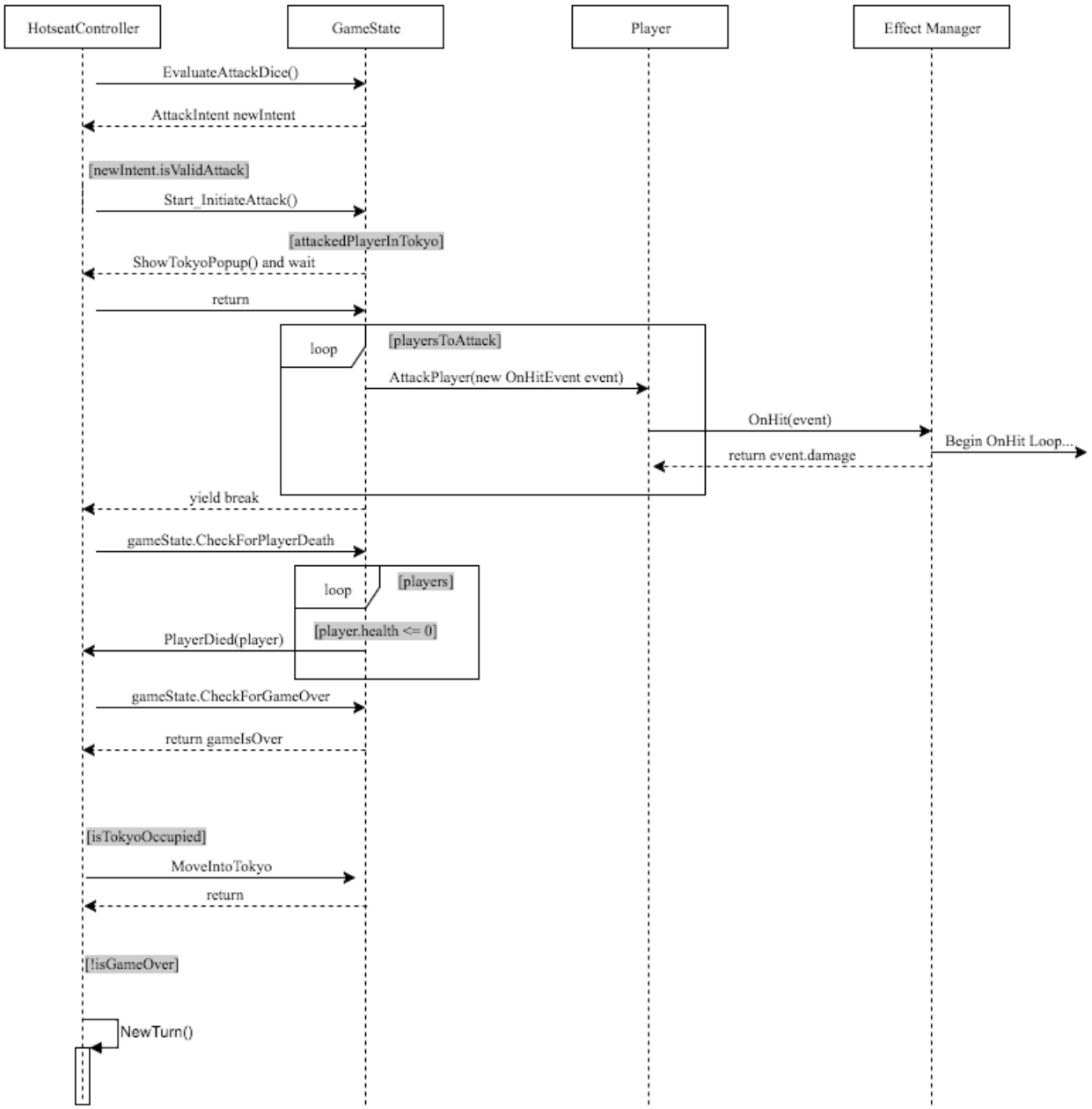

Figure 3. UML Sequence Diagram for end turn

Upon a player ending their turn, a series of coroutines and checks must occur to ensure that proper OnEvents are triggered, if necessary, as well as other checks diagramed above. Once a player's turn has ended, an AttackIntent object is generated that contains information about the attacker and the player(s) being attacked, as well as the damage being done. This allows iteration through the list of players to determine who is under attack, and if certain OnHit events should trigger from the event system from passive card effects, or if a player should be prompted to yield Tokyo to the attacker. Once these checks have occurred in this order, the game logic can properly determine if a player has died (or possibly mitigated the damage through the use of a passive card effect), or trigger the game to end.

In order to ensure that events are handled in the correct order before applying damage, leaving Tokyo, etc, the project utilizes Unity's multithreaded coroutines with yield returns**.** Normally, functions are done synchronously in order. This can cause problems when player input is required to move the game forward; there is no easy way to wait for player input. One solution is to store the state after one function and wait for the player's input before continuing in another. This can become very clunky as more and more effects come into play and need to be stored.

Instead, using thread waiting, functions can pause while another thread is opened and user input is requested. Once the player has given the requisite input and that is complete, the thread continues. This allows for complex logic where one player might be asked information that then triggers another loop asking for information from another. This makes it much easier to control the order in which CardEffectManagers respond to game events; for example, one thread can be paused with an OnHit event while another thread is started responding to the OnHeal event, which finishes before the OnHit event ends.

This system was developed while working on the card "Evasive Maneuvers." If played when attacked, this card allows the player controlling Tokyo to leave without receiving damage... Before implementing this multithreading approach, the player was damaged, the damage amount was stored, and then the damage was removed if they used the card. The developers realized that this type of workaround would have to be performed for every special case as more card effects were implemented. Using multithreading, the function asks the player if they want to leave, and pauses. If they choose to leave, this triggers a separate thread with an OnLeaveTokyo event. In response, Evasive Maneuvers removes them from the AttackIntent (list of players to attack). Then, the original function continues, and because they were removed from the AttackIntent in the other thread, they are never damaged. No workarounds are required, the effects respond to events as they should, and players get to use their hard-won cards.

3. Team Development

Developing software in a team environment has multiple challenges that may hinder progress if not addressed properly. Constant communication among members to ensure their copies of the codebase are up-to-date with one another's is an essential measure to prevent potential time-hindering issues from arising [7]. If a group member retains an out-of-date copy of the project and continues developing on it without being aware of the newest changes made by other members, issues with overlapping code will occur. These issues will result in unintended bugs, unexpected exceptions, merge conflicts causing compile failure and/or requiring time to fix, or causing two or more members to write similar code at the same time, resulting in time wasted.

Utilizing a version control system such as Git [6], and readily available tools to aid collaboration such as GitHub, we are able to easily maintain recorded changes at any time to the project, be aware of the changes other members are committing, as well as manage and track issues using the issue tracker built into GitHub. This helps keep the codebase publicly up-to-date with all members so that the risk of running into issues is minimized and the developmental progress is not impeded.

3.1 Git Branching

Branching in Git is another useful feature that provides aid in the development process. If a new feature is being developed that could possibly interfere with others' work for the time it is being developed, a new branch in the Git repository is created so that the timeline stays separate from the continual, frequent changes being made to the master branch's timeline. Once a feature is finished or ready for deployment it can be merged back into the master branch and any conflicts addressed immediately. Git branching is the solution to working on large changes, or experimental features separately, without potentially introducing bugs, or breaking code in the main, master branch. With the benefits of branching, project members are able to contribute to these changes, while at the same time continue normal development flow in the master branch.

4. View Design

The View Design, as mentioned in previous sections, consists of 3D models and a 2D user-interface. Since the View Design is the section of the game that the user will interact with and see, careful planning and consideration were taken in order to decide on proper design and layout. The visual approach chosen was more simplistic and cartoonish, almost humorous, than complicated and scary. This approach was chosen for the design advantages available for the creation of both the models and user-interface. Unity's ability to import a variety of file types such as images and 3D topology was utilized by creating the models and user-interface in different graphics programs that specialize in 3D and 2D respectively. Due to the differences in designing in 3D and 2D, the work could be divided into different tasks, independent from one another.

4.1 Models

The three dimensional modeling program utilized to create the 3D figures for the project is 3ds Max by Autodesk. This is a well-supported and easy-to-use 3D modeling program that is one of many that are provided for free student use. The idea for each KaijuKO model was agreed upon as a group and then left to the project's modeler and his own creative process. These models were designed with a soft, almost cartoonish style in mind. This simplistic style allowed for the use of fewer polygons which means there are less faces to be rendered which reduces lag and enhances performance in the game environment. The general idea for each model was originally followed, but designs were allowed to grow and change over time. This sped up their construction and fostered more complex designs. To further enhance gameplay each model was further optimized. This involved combining existing polygons into larger faces. Figure 4 below shows the Alien Tank Model after optimization. In total, optimization of all models lowered the games poly-count from 6 million to 1 million faces. These models can use as many as 548 individual pieces to achieve the desired appearance. All models were critiqued by the group before being finalized and implemented in the project.

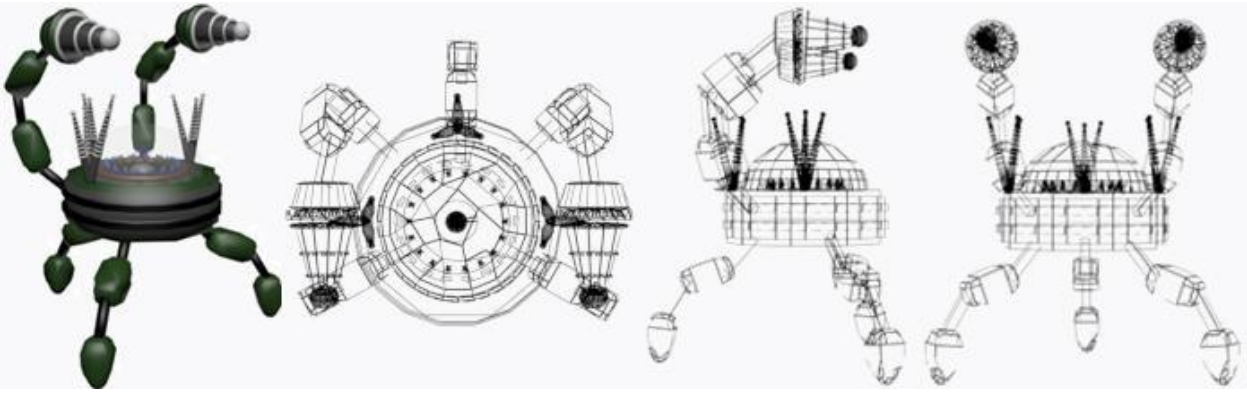

Figure 4. Alien tank model.

4.2 User Interface

The User-Interface, or UI for short, is the section of a game that the players interact with, by displaying important information or taking player inputs [4]. For a quality UI, it must display this information in an aesthetically pleasing manner that matches the overall theme. In the case of KaijuKO, the UI has several different buttons to allow the player to play the game, as well as areas that display information like the player's health and points.

The UI was developed over a course of several steps. First was figuring out what basic information needed to be displayed; such as health, points, and energon; as well as what inputs the game needed from the player; such as roll, play card, and end turn. Once this step was complete, a simple but functional UI was created using built in graphics and tools in Unity, shown in Figure 5a, in order to test the game. Since the game could be tested with the simple UI, mockups were created to get a general idea of what graphics and other elements would be needed. The theme of buildings, in the form of skyscrapers with billboards was chosen due to the game's overall theme of being in a city. It allows for a more immersive experience, as well as a gridded system for the buttons, bars, and other elements to be displayed.

Using the mockup, shown in Figure 5b, graphics were created for the background and buttons. These graphics were created using Adobe Illustrator in order to give them a more simplistic design as well as be in vector graphics. This more simplistic design allows players to easily find the important elements without be distracted, which a more complicated design may have, while still be aesthetically pleasing. The use of vector graphics allows the graphics to be easily scalable, without a problem with pixelating that a raster graphic has, which allows for different screen sizes. After the completion of the graphics a final mockup was created in order to see what a final UI would look like, to ensure the design was correct.

The next step was to import all of the graphics to Unity, to make the planned UI. In order to continue testing other parts of the project, a new scene was created by duplicating the simple but working UI. With the graphics imported and the new scene created, the UI was created using the mockups and tested to ensure that there were no issues. The planned UI was finished, shown in Figure 5c, but had room for changes in the figure, if thought necessary by the team.

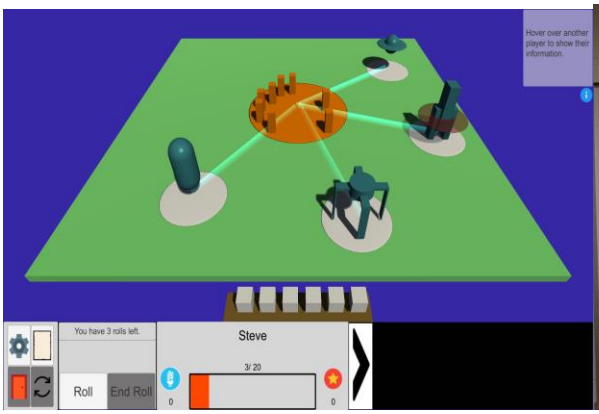

Figure 5a. The Unity Functional UI

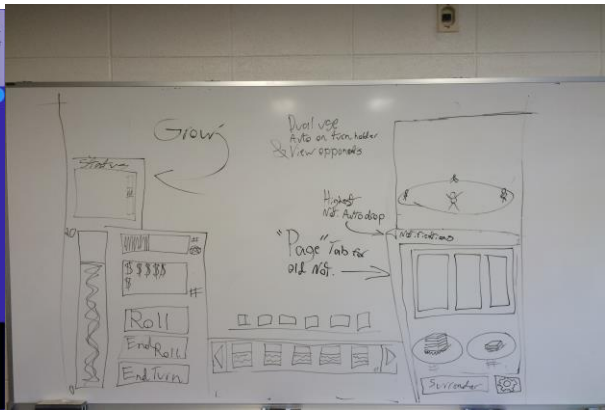

Figure 5b. Mockup drawing for new UI creation

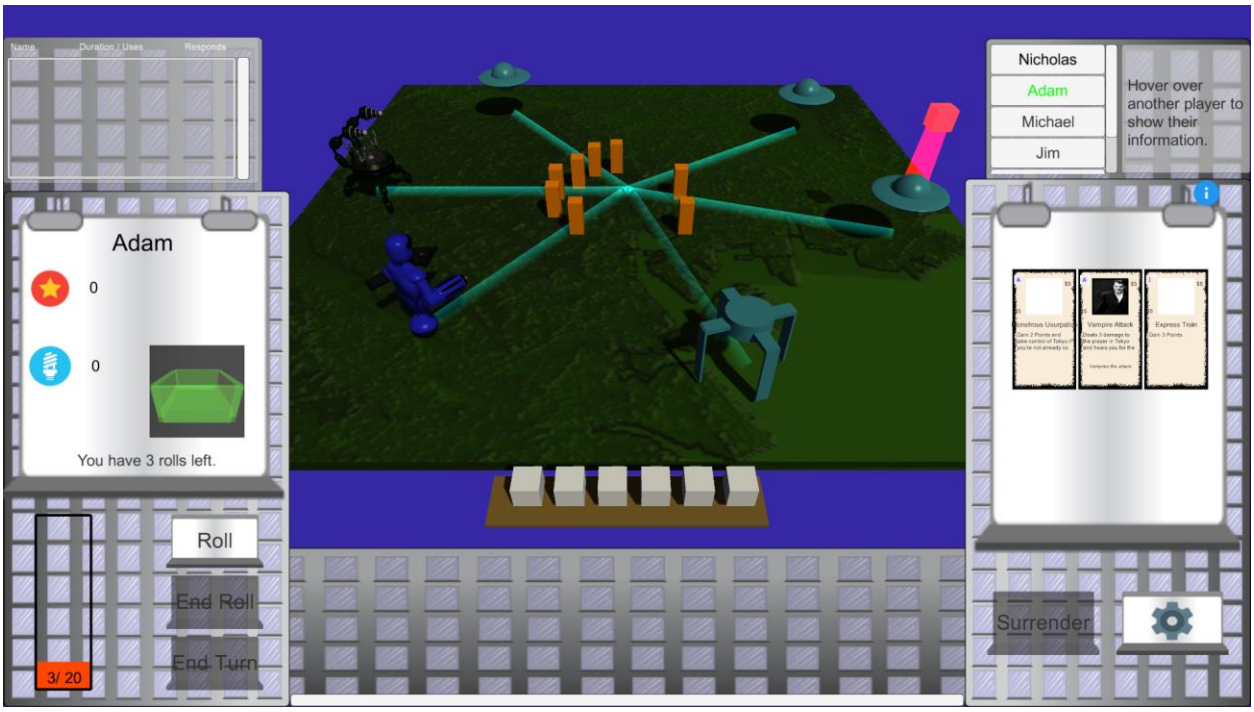

Figure 5c. The new UI, also showing off a couple of the new monster models and board texture

5. Upcoming Features

KaijuKO is a versatile game allowing for nearly unlimited expansion and revision, one of the reasons that it was based on King of Tokyo. Due to the nature of team and research development the project has planned advancements that have not yet been designed.

There are several ways to advance the codebase of the game. Primarily, more cards can be added with effects ranging from simplistic modification of player resources (health, energy, et al.) to more complicated features such as stealing other players' cards. These would entail adding and modifying code in several of the game's state systems, such as the Card Event system. In addition to cards, network multiplayer would allow for people to play the game without needing to be in the same room. Building an AI module would give players a chance to hone their skills in preparation for upcoming tournaments. A replay system would let players preserve their best moments for posterity. Finally, a save / load system would allow the players to put the game down and pick it back up seamlessly.

KaijuKO has a plethora of different art elements that can be designed by the student researchers to better their understanding of graphic design for games. Some examples of these elements are the cards, board, start screen, and sounds. These elements currently have either simple designs, or graphics from Google created to ensure the game could be played and have graphics. These graphics have been added to the future plans to design and change, with the final goal to have all art be original.

6. Conclusion

KaijuKO has come a long way since the first scripts were written back in the spring of 2015. Although there have been some hiccups along the way, the use of good object oriented coding practices has allowed the project to balloon in size and scope without becoming unmanageable. MVC architecture has made working with the code not only easier but much more enjoyable, as the designers can focus on the parts that they need to whilst knowing that they will plug in seamlessly to the greater project. This decoupled design, where team members can build separate portions of the software contemporaneously that then interface with the group's work as a whole, is a useful tactic for completing these large-scale group projects.

7. Acknowledgments

The spring 2016 undergraduate research team would like to express our thanks to the previous group members who laid the original foundations of this project from the previous fall semester. We would like to thank Sean Stamm and James Morrow for their excellent contribution and software design that left us a great framework to continue to build upon and expand. We would also like to extend our thanks to our faculty advisor, Dr. Adam Whitley for his guidance, encouragement and passion on this project.

8. References

1. Unity Technologies, Unity, 2015, https://unity3d.com/unity, retrieved Sept. 24th 2015
2. La Voie, M., "Is there a performance difference between Unity's Javascript and C#?", Unity, 2009, http://answers.unity3d.com/questions/7567/is-there-a-performance-difference-between-unitys-j.html, retrieved Sept. 21st 2015
3. West, M., "Evolve Your Hierarchy", Cowboy Programming, 2007, http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/, retrieved Sept 21st, 2015.
4. Quintans, Desi, "Game UI By Example: A Crash Course in the Good and the Bad", envatotuts, 2013, http://gamedevelopment.tutsplus.com/tutorials/game-ui-by-example-a-crash-course-in-the-good-and-the-bad--gamedev-3943, retrieved March 16th 2016
5. Lambert, Steven, "Intro to Object-Oriented Programming for Game Development", tutsplus, 2012, http://gamedevelopment.tutsplus.com/tutorials/quick-tip-intro-to-object-oriented-programming-for-game-development--gamedev-1805, retrieved March 16th 2016
6. Git. Accessed April 21, 2016. http://git-scm.com. Git version control software
7. O'Sullivan, Bryan. "Making Sense of Revision-control Systems." Communications of the ACM, September 2009, 56-62.