Card Tricks: A Workflow for Scalability and Dynamic Content Creation Using Paper2D and Unreal Engine 4

Extended Abstract

Owen Gottlieb
Rochester Institute of Technology
Rochester, NY
oagigm@rit.edu

Dakota Herold
Rochester Institute of Technology
Rochester, NY
dxh8497@rit.edu

Edward Amidon
Rochester Institute of Technology
Rochester, NY
eha8618@rit.edu

ABSTRACT

In this paper, we describe the design and technological methods of our dynamic sprite system in *Lost & Found*, a table-top-to-mobile card game designed to improve literacy regarding prosocial aspects of religious legal systems, specifically, collaboration and cooperation. Harnessing the capabilities of Unreal Engine's Paper2D system, we created a dynamic content creation pipeline that empowered our game designers so that they could rapidly iterate on the game's systems and balance externally from the engine. Utilizing the Unreal Blueprint component system we were also able to modularize each actor during runtime as data may be changed. The technological approach behind *Lost & Found* uses Unreal Engine and Paper2D in order to maximize scalability and dynamic content creation. We believe our methods will be useful for any developer with large volumes of data, intensive procedural content in their game, or those who would like to improve their workflow when working with dynamic data.

Keywords

Games and Learning, Scalability, Dynamic Content Generation, Unreal Engine, Paper2D

1 INTRODUCTION

The ability to modify game elements is critical for designers, so designers must often build modular systems with this modification criterion in mind. Given the ever-increasing demands for content volume and change, developers must find new ways to create and scale content. Over the course of the production of *Lost & Found* we created a workflow in and around Unreal Engine 4 to meet the content needs of our game. In doing so, we believe that we have found ways to take advantage of the Paper2D system that will allow other designers with related goals to also create effective dynamic data-oriented systems that facilitate developer-generated content. This method can work for both porting completed card games to mobile and also for iterative design of mobile-based card games. The value in terms of iteration can be seen in circumstances such as re-balancing a game.

As mobile games shift and advance based on user feedback and testing, this method can allow for fast-response content generation. We first provide an overview of the game to provide specific context. In the sections that follow, we provide a description of each component of our system, how each component relates to our architecture, problems we faced with each component, our solutions, and suggest how these solutions might be applied by other developers and designers for related systems.

2 GAME OVERVIEW

Lost & Found is a digital mobile prototype (funded by the National Endowment for the Humanities) of a card game that teaches medieval religious legal systems, beginning with Maimonides' *Mishneh Torah*, with plans to expand to related contemporaneous Islamic law. The game is set in Fustat (Old Cairo) in the 12th Century with attention to historical detail and accuracy. Players must balance the needs of their family with those of the community, and the game explores collaborative, oft-overlooked prosocial aspects of religious legal systems. The overall learning goal of the game is to widen discourse around comparative religion through understanding legal history. The game progresses across the four seasons of the year, in which each season contains events that help or threaten the community. The events revolve around the laws in the *Mishneh Torah* regarding lost and found objects. In order to win, players must learn to work together to complete "communal" responsibilities while attempting to simultaneously complete a set of individual "family" responsibilities. Players choose how to help themselves and others, making for a unique dynamic, resulting in either a win of any number of players, a no-win state or a communal loss. Communal loss state is triggered by any player going "destitute" during the game, when they face a resource deficit so high that they cannot complete their turn. The game contains a variety of kinds of decks of cards such as events, resources, and lost resources. Each deck has a set of values and various effects on gameplay. The task of translating the card game to mobile presents unique challenges for this method.

3 DATA CREATION FOR ACTORS: FROM SPREADSHEET TO ENGINE

In the early stages of planning for *Lost & Found*, we sought to design a system that organized data in a way that would be convenient to both read and manipulate. This was important, because the game contains 164 cards and we also had to adhere to

the various platform constraints of mobile. Even though there are many variations of cards, we define a "card" as singular piece of data with a set of key-value pairs. Keys are different fields such as "name, type, and owner." Values are the information associated with that key such as "Garment, Starting Resource, and Cowherd." An "actor" is an Unreal Engine term for an object that can be placed or spawned in a level[1]. The game designers used an excel spreadsheet to organize all of the information for each of the cards in the game. Maintaining use of the spreadsheet was important for the designers because it was the most efficient way for them to allow for editing and iteration on the cards in future versions. The technical advantages of using a spreadsheet include that it is particularly helpful for serialization in the way that we described a "card" above.

The choice of Unreal Engine was determined by a number of criteria, including: it would deploy to both iOS and Android natively, it has robust built-in networking capabilities, and perhaps most importantly, that the pro-licensing EULA was within the project budget. Once Unreal Engine was locked in, we would have to develop a pipeline for it. We believe others will find themselves facing similar cost and capability constraints, and therefore can benefit from this workflow.

We designed a solution that would allow designers dexterity in manipulation of the card data that feeds into the game. While Unreal supports CSV and JSON importation, types must be predefined through a data table within the engine. This was not suitable, because we needed a system that could handle changing data types at the discretion of the designers. We achieved this through the use of an external card parsing tool we developed in C#. We output our data from this tool in two formats, JSON for debugging our parser and a CPP file that contained a TCHAR array for use in C++ [3]. This tool allowed for our entire spreadsheet to be encapsulated in a single, lightweight class that accounted for changing types and could easily be reverted to JSON for validation. This process could be repeated in a few clicks for trivial manipulation and iteration on the cards in the spreadsheet.

Using the translated data from the parser tool, we could then recreate the JSON objects that the engine could understand. From this point, we expanded on the serialization structure. We organized the cards into arrays inside the parser through formatting. In this way, we sorted the cards into decks and differentiated them by type. Even though the engine could understand both the data and how it was organized, we still had to store serialized card data in a data structure that Unreal used. A UObject is the most basic class that can be represented inside of Unreal. Inside of each UObject we created for each card, we assigned properties which come from the JSON data itself; however these remained dynamic through the use of unions.

Once the data was a UObject, we could output information from each card (such as names) to the screen as debug output, in the engine. Another technique we used, which would help us later was serializing the image names. We did this because we needed a way to know which image was needed for each card. Using the properties in the card allowed us to subsequently build the visual representations of the cards using sprites in Blueprint. "Blueprint" is a visual scripting system in Unreal that contains almost all of the functionality of Unreal's version of C++. An important exception would be attempting to inherit from a Blueprint class to a C++ class [2]. Parsing the data into a UObject built a basic object with properties that designers, programmers, and the engine could easily understand. Having access to this data at each stage of the pipeline, from spreadsheet to JSON to UObject, allowed us both more flexibility as well as the ability to constantly check for errors.

As we designed this workflow, we always focused on where the data was going as well as how we could streamline and simplify the process for others who would join the project so that they could understand how to make changes. For designers working in card or other strategy games using data from spreadsheets, this conversion method can be particularly helpful because it decouples design, balance, and iteration from direct programming implementation. In later sections, we expand on how we use our UObject to begin creating actors that would be visible in the scene that could be changed in C++ and Blueprint.

4 PAPER2D: UNDERSTANDING THE SPRITE SYSTEM FOR UNREAL ENGINE

Paper2D is a plug-in that is packaged with the Unreal Engine. It is the primary sprite system used for creating 2D games in Unreal. The system is intuitive to learn and can translate imported images into sprites through simple drag and drop. In order to understand how we assemble the cards from our data, it is important to first highlight the plug-in itself by explaining its benefits and drawbacks in the context of this project. For the context of this section, an "assembled card" refers to a group of sprites containing all of the images that make up a card. See Figure 3.

Because this project is targeted for mobile devices, the default importation of images resulted in sprites that were uncompressed; while this provides high quality images, the file sizes were too large given our platform. In order to address this issue, we compressed the raw images into sprite sheets and imported the sheets instead. This reduced the overall file size by half from 30MB uncompressed to 15MB and maintained an acceptable quality. This was critical given the 307 images in the game and their centrality and importance in the game. Using this workflow, when we received a new piece of art from the illustrators, we had to reimport the entire sprite sheet as opposed to simply loading in just the new sprite. This is the case because we were using the packed atlases. While time consuming, this approach provided greater benefits than using the uncompressed images when comparing executable file sizes. The executable with packed images from TexturePacker was considerably smaller, which is critical for mobile app stores. Another benefit of Paper2D is that it

includes a great deal of compression and image optimization functionality. This was most important in *Lost & Found* when trying to sharpen the UI sprites in the game. Changing the import settings on the sprite to 'UI' made those sprites render far more sharply than before. Because those sprites were constantly on-screen, the overhead in rendering them was negligible compared to when we tried rendering assembled cards as UI sprites. While we found this feature of Paper2D beneficial for user interface sprites, it was not useful for the cards themselves because UI sprites are always in the front of the rendering queue. The fact that those sprites are always in the front of the queue caused visual problems to the player when cards had to overlap. Therefore, we needed a new solution to account for the complexity of rendering assembled cards on screen.

One challenge we faced with Paper2D was the rendering complexity of assembled cards. Because Unreal Engine is primarily a 3D engine, its material system is focused on 3D meshes. A material can be best thought of as the "paint" that is applied to a mesh. Even though we are not using 3D objects, Paper2D includes default materials for sprites. The default materials were used without problems, but only until we added card back sprites to the cards. Then each sprite on the front of the card continued to render on the back, but then, the player could see sprites reversed when cards were flipped on the vertical axis. It also doubled the required rendering cost as sprites only need to be seen from one direction which is when they are facing the camera. We solved this problem by writing a material shader that only rendered sprites on one side facing the camera which cut the rendering load in half (by only showing what was immediately visible to the player). This solution provided noticeable performance improvements in game in terms of play speed. The front and back sides of our material can be seen in Figure 2 and Figure 2.1. Outside of this one issue, Paper2D caused us no problems in terms of image quality or performance. Importing sprite sheets and swapping sprites during runtime was clean. Paper2D is lightweight and provided enough flexibility to load each piece of a card in, rather than a single flat image. Working with Paper2D can allow for an extensible 2D sprite system that we found helpful in adapting to various design needs.

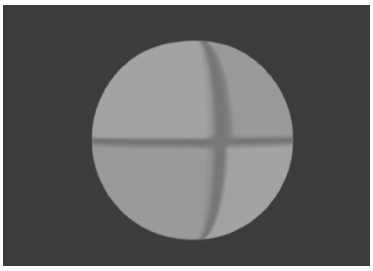

Figure 2: Front side of our card material.

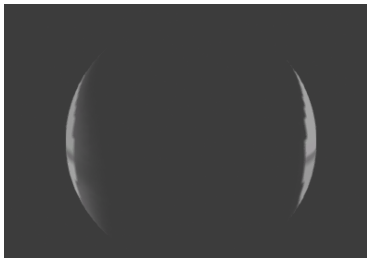

Figure 2.1: Back side of our card material.

5 AUTOMATED ACTOR GENERATION: THE DESIGN AND IMPLEMENTATION OF CARDBP

Often in games, that which is procedurally generated are game *environments.* These environments have limited player interaction. In our case, we designed procedurally generated objects with which the player continually interacts. Here, we describe how we procedurally generated our card actors.

Upon designing and translating the spreadsheet data into an engine readable object, we had to design a system that could translate that data into an actor in the scene. This section describes how we melded card data and sprite images into a cohesive entity. We began by extending the card UObject that we created from our data earlier to an actor class in C++ that contained all of the variables from our data. Because it is beneficial to visually see the cards while we're working with them, we further extended our card actor class to a Blueprint which we named CardBP. All of our gameplay scripting for transforming cards is also handled by CardBP. This allowed us to rotate, resize, and linearly interpolate CardBPs in the scene during the game.

CardBP holds a reference to the card data as well as Paper2D component sprite slots. Each card in the game has a basic set of properties that fill these slots such as a frame, background image, category, cost, and card background image. The images are stored and imported into the engine separately, although the card data contains all of the image names. However, CardBP also contains all of the possible specific sprite slots that may be needed by a card regardless of whether or not that card uses that slot. Examples include, "BonusToCategory," "BonusFromCategory," "Cost," etc. The "Train A Doctor" community responsibility card in Figure 3 illustrates the use of these additional sprite slots. When a CardBP is created, all of the sprites are left blank so if the card data indicates that CardBP does not contain those sprites, then they are not generated. There was very little overhead for keeping unused sprite components because there are fewer draw calls based on the images on screen. CardBP served as the base Blueprint class that we used to generate our dynamic cards from the JSON data. The methods for CardBP can serve as an example of how our workflow can serve as a model for those who would also like to build adaptive actors inside Unreal Engine 4.

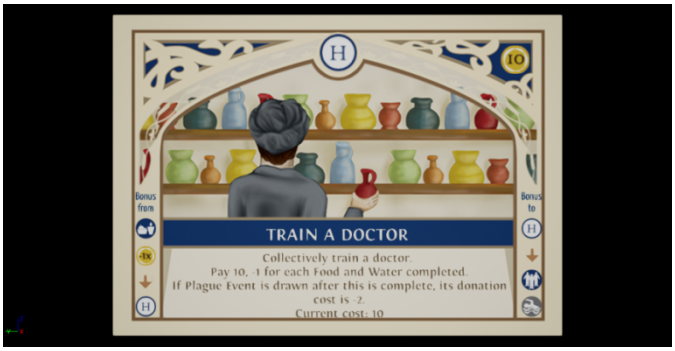

Figure 3: An instance of CardBP in the editor.

5.1 Organizing the Sprites

Because *Lost & Found* is a table-top card game, the conversion of the game to a digital medium meant the volume of images required to display the 164 cards was substantial. We had to reduce the image file sizes. Our images were delivered from our illustrators in .PNG format and compiled into an atlas of sprites in the TexturePacker program. The atlas file essentially tells Unreal which section in the sprite sheet is an image, acting as a kind of cookie cutter in a wide swath of cookie dough. Unreal then translates each image into a Paper2D sprite, which we could use in the game. All sprites followed a specific naming convention and were then placed in one folder. From this point, whenever a CardBP needed to populate a sprite component, we retrieved the path to the folder and appended the sprite image name from the card's data. We then created a function in CardBP to load the sprite asset into the component based on that path. By organizing and creating these atlases of sprites, TexturePacker allowed us to pull the packed images from only 4 atlases instead of 307 individual image files. This, in turn, significantly enhanced ease of maintenance for our system.

5.2 Runtime Generation

The most flexible part of our system is runtime generation and manipulation of Paper2D sprite components. When a CardBP is first placed or spawned in the world, it generates all of the required sprites based on the card's data. CardBP then offsets each sprite based on the type of card that the sprite represents. For example, the value sprite on a resource card has a different position in local space than the cost sprite on a communal responsibility card. CardBP accounts for all of the nuanced offsets of each type of card to best replicate the card's print counterpart. CardBP also does this with font styles, colors, and sizes for the card's text through Unreal's text render components. We are also able to modify and reload any of these during runtime. This is most notable when a player pays toward the cost of a communal responsibility, because the cost sprite on the card will decrease as it is paid down by the players. This was particularly helpful for players as it offloaded mental arithmetic required in the print version of the game to the machine, taking advantage of the mobile format. The generation of the CardBP is important as it allows an almost instantaneous change in information shown to the player by manipulating the card properties, and reloading the changes to the sprites. Developers can follow this model in order to provide real-time feedback to players.

6 CONCLUSIONS

We began this project with key game mechanic, technical, and business requirements based on our card-strategy game. We sought to develop a streamlined method for designers to be able to easily manipulate the game's data as well as a system that would provide a manageable learning curve for onboarding new team members – all cross-platform, and within a manageable budget.

With the primary data in spreadsheet form, we built a card parser to transfer the data to JSON. This transfer allowed us to maintain data manipulation functionality, as it kept the data in human-readable format. Next, we turned the card data into a TCHAR array which we loaded into Unreal using C++. Then we created a UObject in C++ from the TCHAR array data in order to build a card actor (CardBP). Leveraging Paper2D, we created a material shader that optimized the rendering complexity of the image sprites used by our cards. We further optimized our images by compressing them with TexturePacker and using atlases (as opposed to using singular images). We were then able use CardBP, a template, to load-in and offset sprites based on the card's specific data. This approach provided us with a pipeline for dynamic content generation that was scalable and would allow us to generate a high volume of cards with specific data. This reduced rendering load significantly. It also allowed us to continue to iterate data in any future variations

For developers working on data-oriented game genres such as CCGs, TCGs, or strategy games, creating scalable solutions for dynamic content generation is important. We believe that using this kind of workflow, one that emphasizes the design of the data in addition to the programming implementation, can make for an efficient iteration cycle. We hope that providing this case example can serve as a model for approaching data, one that will facilitate a wide variety of approaches and applications by game developers and researchers in the future.

ACKNOWLEDGMENTS

This work is supported by the GCCIS, Office of the Vice President for Research, and the MAGIC Center at RIT. This work is also supported and funded by the National Endowment for the Humanities. Any views, findings, conclusions, or recommendations expressed in this paper do not necessarily represent those of the National Endowment for the Humanities.

REFERENCES

[1] Unreal Engine 4 Actor https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/

[2] C++ and Blueprints https://docs.unrealengine.com/latest/INT/Gameplay/ClassCreation/CodeAndBlueprints/

[3] TCHAR typedef https://msdn.microsoft.com/en-us/library/office/cc842072.aspx?f=255&MSPPError=-2147217396