ScriptEase II: Platform Independent Story Creation Using High-Level Patterns

Kevin Schenk[1], Adel Lari[1], Matthew Church[1], Eric Graves[1]
Jason Duncan[1], Robin Miller[1], Neesha Desai[1], Richard Zhao[1]
Duane Szafron[1], Mike Carbonaro[2] and **Jonathan Schaeffer[1]**

[1]Department of Computing Science, University of Alberta, Edmonton, AB, Canada T6G 2E8
[2]Department of Educational Psychology, University of Alberta, Edmonton, AB, Canada T6G 2G5
{kschenk, lari, mfchurch, graves, jtduncan, remiller, neesha, rxzhao, dszafron, mcarbona, jonathan}@ualberta.ca

Abstract

As the video game industry grows, both developers and creative authors seek new ways to simplify the process of controlling story content using scripts. This paper describes a story model and its software implementation, ScriptEase II, designed to solve this game design bottleneck. ScriptEase II is the second generation of the ScriptEase system, whose goal was to enable game authors with no programming ability to generate scripting code from high-level game patterns. ScriptEase II differs from the original in two important ways. First, ScriptEase II uses game-dependent translators to generate scripts for any game engine. Second, ScriptEase II uses a drag-and-drop interface that simplifies the story component creation menus that grew cumbersome in the original ScriptEase. The feasibility of code generation has been validated using three different game engines and the advantages of the simple drag-and-drop interface have been validated by a user study.

Introduction

Game design tools that are able to quickly and reliably add procedural content are a major research area, and will become a major target of investment for many studios. These tools will allow authors to create and integrate content. Much of the work in story-based game development focuses on narrative content creation, which includes intricately connected elements such as storyline, subplots, characters, quests, and their interactions. Converting story content "into the program code necessary to create the desired behavior" (Cutumisu et al. 2007) is a bottleneck in content generation. As Cutumisu *et al.* posit, allowing authors to implement their stories into gameplay without the help of programmers would remove this bottleneck and ease story creation.

ScriptEase II (referred to as SEII) is a game engine independent tool written in Java that uses high-level game patterns to automatically generate scripting code. It replaces manual scripting with a drag-and-drop interface that simplifies the creation of complex stories. Similar visual code generation tools are available, such as Alice (Cooper, Dann, and Pausch 2000), and Unreal's Kismet (Unreal 2011). However, these tools do not support the same affordances as a tool able to utilize high-level story patterns like SEII or the original ScriptEase (referred to as SEI hereafter) (Cutumisu et al. 2007). Unlike SEI, which only targeted BioWare's NWScript for the *Neverwinter Nights (NWN)* game engine (BioWare 2002), SEII can target any game engine or API as long as a translator is created for it. SEII also features a simpler user interface, which further enhances the content generation experience. The ease of SEI was verified in experiments by Carbonaro *et al.* (2010), where they found that 10th graders with no programming experience were able to successfully produce interactive stories. The goal of script generation research projects, such as ScriptEase, is to allow designers to generate scripts using a small complement of concepts. SEII is based on five simple story components: Story Points, Causes, Effects, Descriptions, and Controls. These pattern categories can be abstracted across most event-based games. Cutumisu *et al.* (2007), and Trenton *et al.* (2010) have shown the effectiveness of using high-level story patterns to aid story creation.

SEII has three distinct target audiences. First, it can serve as an exemplar for how game companies can create similar tools for their designers who are not programmers. Second, it can be used by both independent game designers and novices who are learning to design games. Third, it can be used by academics and game companies to study the use of AI in computer games (Zhao and Szafron 2009).

Related Work

Translating ideas into video game code is a challenge, especially for those with limited programming experience. Multiple studios and researchers have tried to solve this problem by creating tools that aid the creation of scripting code based on a set of patterns or simple building blocks.

Alice is an educational tool developed at Carnegie Mellon University. Users are able to generate scripts for Alice's 3D world by combining graphical tiles to form instructions and animate 3D objects. Manipulating objects in this way generates code, and teaches users computing and programming concepts (Cooper, Dann, and Pausch 2000).

Kismet is a tool in Epic's Unreal Development Kit (Unreal 2011) that provides a graphical interface for script creation. Since Kismet is designed for professional game de-

velopment rather than being an educational tool, it allows certain affordances that Alice does not. Users are able to connect various actions and conditions to create sequences. They then use an annotative flow chart to select appropriate behaviors that occur when an event is fired on an object. While Kismet allows for some complex interaction in its environments, the sequences are generally not reusable and the low level focus makes it difficult to learn and use (Cutumisu et al. 2007). Additionally, Kismet is proprietary software and thus limited to the Unreal Development Kit.

Other examples include MIT's Scratch (Resnick et al. 2009), WeQuest (Macvean et al. 2011), Skorupski and Mateas' Story Canvas (2010), Pizzi and Cavazza's Interactive Storytelling tool (2008) and Microsoft's Kodu (MacLaurin 2009) plus extensions (Fristoe et al. 2011). Each tool is tied to a single specific game engine.

From Original ScriptEase to ScriptEase II

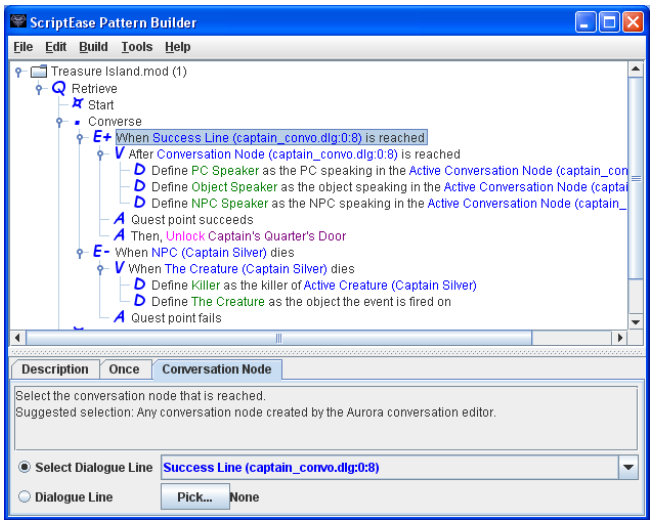

Figure 1: A simple story pattern in the original ScriptEase.

The two main goals of SEI were a) to have all authoring done with high-level story patterns and b) to present the authors with a natural description of the story they wished to create (Cutumisu et al. 2007). SEII retains these goals, but implements them with a drag-and-drop interface as opposed to SEI's menu-driven approach. When comparing a story created with SEI from Figure 1 with an equivalent story created in SEII from Figure 3, it can be seen that SEII provides the same natural language descriptions but with a more visual interface.

While SEI uses a text-based view of a story in an indented tree-like structure, SEII uses a graph structure to show the story within the context of the game. Each story graph node (called a Story Point) contains related cause-effect story components. Since a story in a non-linear story-based game is analogous to a decision graph (Trenton et al. 2010), SEII provides a view that is more visually consistent with the story model. SEII is also game independent. Whereas SEI can only create stories for BioWare's *Neverwinter Nights* engine, SEII can target any game engine or API for which a translator is built.

In SEI, stories are created using patterns selected from a set of menus. To create the story in SEII, cause-effect story components are dragged from the Library pane on the top left of the screen to the Story Point pane on the bottom right. Objects from the game-object pane on the bottom left are then dragged onto the story components to contextualize them for the story being told. Since translators may contain over 100 story components in their library, SEII also provides a search filter to further simplify finding and selecting appropriate cause-effect story components.

ScriptEase II Story Model and Story Creation

SEII generates scripting code and attaches the scripts to game objects. We refer to the entire collection of scripts attached to a game as a *story* and the user as a story *author*. SEII aims to be accessible to a larger demographic, with users as young as elementary school students. Carbonaro et al. (2010) found that the use of high-level patterns alone allowed tenth graders with no computing skills to make fairly intricate stories without programming or scripting. As stated by Trenton et al. (2010), "game patterns that describe basic game interactions are usable by the general population without the knowledge of computer scripting."

The Story Graph and Story Points

In our model, a story is represented as a directed acyclic graph of *story points*, as shown by the simple illustrative pirate story in Figure 2. In real game stories, there are potentially hundreds or thousands of story points. Each story point corresponds to an important event that may occur during the story. As most story-based video games are event-based, events provide a simple and effective way to decompose a story into story points. In our model, each event consists of a *cause* and its associated *effects* that are based on a library of commonly occurring tropes within video games. For example,

Cause: when a lever is pulled
Effects: a door opens
a sound is played

When a story starts, the Start story point is *active*. Until the story ends, one or more story points is *active*. A story point is *visited* when it is active and its cause occurs or has occurred. For example, in Figure 3, when the story starts, the Start story point is active, but it is not visited until the player actually talks to the Captain and reaches a particular dialogue line. When a story point is visited it can either *succeed*, *fail* or *neither* (mechanism explained later). If it succeeds, then all immediate successor story points become active (except for merge story points, also explained later). The succeeded (or failed) story point is *deactivated*. If neither happens, the story point remains active and can be visited again later. For example, if the Start story point in Figure 2 succeeds then it is deactivated and three story points following it are activated: Parrot, Rum and Kill. When a story point's success activates multiple story points, we say the story has

branched. Branching is common in interactive games since the player can follow multiple story arcs concurrently.

If a cause occurs and its story point is not active then the story point is not considered visited. For example, imagine a quest that is initiated after the Captain asks the player to fetch his parrot. If the player fetches the parrot before talking to the Captain, the Parrot story point is not visited and does not succeed or fail at that time. This is important since it is common to have a quest journal entry appear in a player's journal when a story point succeeds and it would be awkward if a quest entry appeared before a quest actually started. What happens if the player fetches the parrot and then talks to the Captain? In this case, when the player talks to the Captain, the Start story point would succeed and the Parrot story point would be activated, as normal. However, there are two possible semantics for out of order actions. We can either remember that the cause has occurred and immediately succeed the Parrot story point or we can wait until the cause occurs again. With the second semantics, the player would have to fetch the parrot again. We support the first semantics since it is the most common one. Using the second semantics or allowing the author to select the semantics would be an easy change to SEII.

It is often possible to follow multiple branches in a story concurrently. For example, in the pirate story, the player may decide to fetch the captain's parrot and then fetch the captain's rum. Another alternative is to fetch the captain's parrot and then kill the captain. When a story branches, the branches often rejoin at a *merge* story point that has more than one predecessor. For example, in Figure 2, the Return story point is a merge story point with two predecessors, Parrot and Rum. Sometimes an author wants a merge story point to become active when any one of its predecessor story points succeeds and sometimes an author wants all predecessor story points to succeed before a story point becomes active. In SEII the author can select how many predecessors must succeed before a merge story point becomes active. For example, in Figure 2, both Parrot and Rum must succeed since there is a "2" in the merge story point labeled Return. If the author required the player to fetch either the Parrot or the Rum, the author would use a "1". It is possible to express any complex requirement on particular story points using this mechanism. For example, if the author wants all three from a particular set or both from another set of two then one intermediate story point is created using a "3" to merge the three and another story point with a "2" is used to merge the two and then a final story point is used to merge the two intermediate story points with a "1".

Creating game content that may not be reached in a single play-through can be expensive for designers. To reduce costs, this extra content is often not implemented by game studios. Lowering the cost of creating branching content allows authors to build more complex stories and increase replay value.

Story Components and Game Objects

Figure 3 is a screenshot from SEII showing a story called Treasure Island. This is the same story as in Figure 2. The story graph is shown in the story pane (upper right). The tool palette on the left edge of this pane can be used to edit the story graph, by adding and deleting arcs and story points. Each story uses some *scene files* that were created by designers using a game-dependent tool, such as the NWN Aurora Toolset or the Unity editor. Each scene file contains game objects, some of which may have been originally created using other tools such as 3ds Max or Maya. Figure 3 uses a NWN scene file (called a module). The game objects are shown in the game object pane (lower left). Examples are a Placeable called Armoire and a Door called Captain's Quarters Door.

If a story point is selected in the story pane, its contents are shown in the story point pane (lower right). Each story point should contain one or more causes. In Figure 3, the "(SP) Start – Talk to the Captain" story point contains the cause "When (DL) <A'right, be right back.> is reached".

Each cause includes a <Subject>, which is used to specify the game object for which the cause must occur. In Figure 3, the <Subject> slot is filled with a dialogue line object (DL). When this particular dialogue line of Captain Silver is reached in a conversation, the cause occurs. To place this cause in the story point, the author first selected the story point in the story pane. Next, the author dragged the cause "When (DL) <Subject> is reached" from the library pane (upper left) into the story point pane (lower right). Finally, the author dragged the "(DL) A'right, be right back" dialogue line object from the game object pane (lower left) onto the <Subject> slot of the cause.

Slots, like the<Subject> one, appear in many story components and must be filled, usually by game objects. All slots specify the types of game objects that are allowed, such as dialogue lines (DL), creatures (C), text (T), or lists (Li). The circles on the left side of a slot indicate the allowed types. A single slot may allow game objects of multiple types. In Figure 3, the <object> being unlocked could be either be a door (D) or a placeable (P).

Each cause can contain *effects*, *descriptions* and *controls*. In Figure 3, an "Animate subject to (Li) [Animation]" *effect* was dragged from the library pane into the cause. The animation "Looping Talk Normal" was selected from the [Animation] list of the effect.

In addition to game objects, the author can also use predefined *implicit objects*. Consider the cause: "When (D)(P) <subject> is closed by (C) Last Closer", which is highlighted in the Library pane (upper left in Figure 3). If this cause was dragged into the story point pane, the "(C) Last Closer" implicit object could be used for any effect that requires a Creature (C). An author can apply effects to the specific creature that closed the door, even though this creature is not preset but dynamically determined during gameplay.

There are other situations where an author cannot identify a particular game object before the game is played, such as the nearest object to the player character, or whether a particular story point is currently active. SEII solves this problem by using descriptions. A *description* describes a game object dynamically and allows the author to refer to the described object with a label. Figure 3 shows an example *description*, labeled "Is Active", which represents whether the "(SP) Start – Talk to the Captain" story point is active or not.

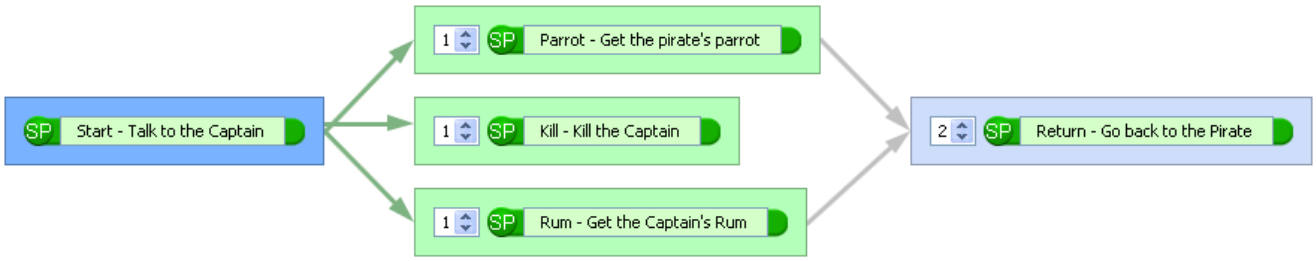

Figure 2: A story involving a pirate and the various story points that need to be reached in order to complete it.

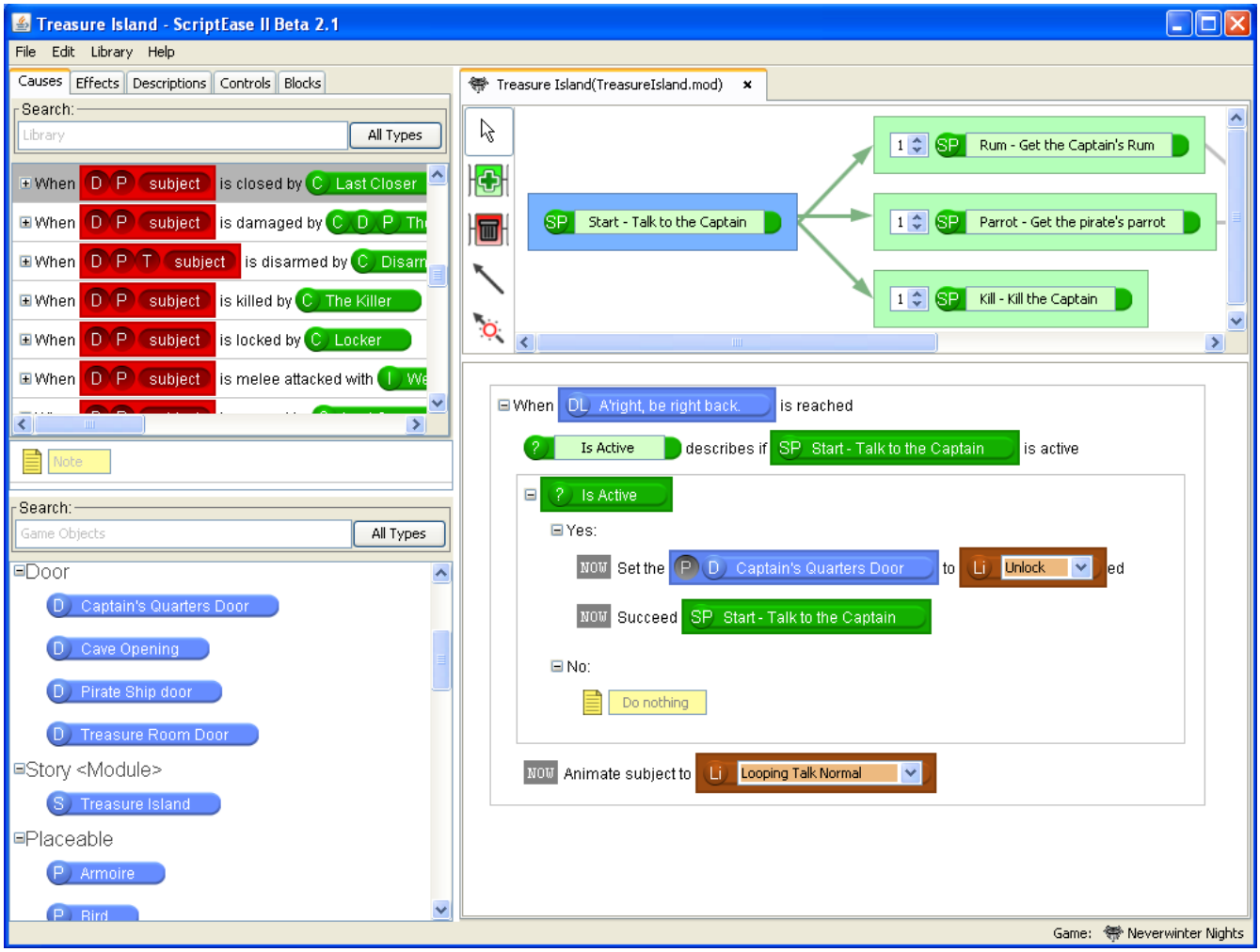

Figure 3: Part of a ScriptEase II story. The top right pane contains the story graph with multiple story points. The bottom right pane contains a single cause with multiple effects for the selected story point.

This description has two possible values, “Yes” and “No”. Description labels can be dragged to slots like game objects or implicit objects. In general, descriptions can be dragged from the library pane into the story pane and then the slots can be filled with objects.

Game stories are interactive so the story depends on player choices. SEII provides a *control* called a *question*, which can be used to control the story. Each *question* has two sections, “Yes” and “No”, that can each contain effects, descriptions and controls that are performed based on the dynamically computed answer to the question. For example, in Figure 3, the “Set the (D) Captain’s Quarters Door to (Li) Unlocked” effect and the “Succeed (SP) Start – Talk to the Captain” effect will occur if the label “Is Active” has value

"Yes" and won't occur if the value is "No". It is very common for authors to want different effects to occur based on whether a story point is active or inactive. Because of this, when a clause is dragged from the library pane to the story point, SEII automatically adds the "Is Active" description for the current story point and adds the "Is Active?" question. The author can simply delete these components if they are not required.

There are currently two other controls in SEII. The first delays the performance of its contents for a certain time (e.g. wait four seconds and then spawn a creature) and the second repeats its contents a certain number of times (e.g. spawn four creature).

Code Generation

To maintain game independence, SEII uses pluggable game translators, where each translator generates scripting code for a particular game engine. Each translator has three components. The first is a set of XML files that define the game engine API. The second is an XML file that specifies the grammar of the scripting language. The third is the implementation of an interface for reading and writing game objects from game-specific scene files. A translator creation tool is available to help developers build translators.

SEII's game independence was proven by creating three translators. The first was for the *Neverwinter Nights* game engine. This translator was used successfully in an undergraduate game design course. A SEII translator was also created to generate C++ code for controlling the scoring system of a physical pinball machine which had a C++ API (Wong et al. 2010). Finally, a translator for the Unity 4 game engine (Unity Technologies 2013) was created. We have used the Unity translator to author a 2D space shooter and a simple 3D RPG.

Translator Creation

Translator Format

A SEII translator requires a Java interface between the three translator components and SEII. The three components are: an API dictionary serving as a library of story components, a language dictionary for the syntax of generated code, and an implementation of the story system. The interface extends a class with specific methods. This interface parses the game file so SEII can find game objects and attach generated scripts to them. The API dictionary contains definitions of the story components and the code to be generated for them. The language dictionary defines the format of the types of story components. Creating a translator will require a programmer to write an implementation of the story system in native game code that can then be used by the effects to control the flow of the story.

Development Time

The time required to write a translator depends on the complexity and documentation of the game file format, and the programmer's experience with the game scripting code. For example, the *Neverwinter Nights* translator took months to develop due to the complex format of the scene files and the fact that this was the first translator written for SEII. Conversely, the Unity translator was generating code after just two weeks of development, because its game file format is written in plain text YAML code and it was the third translator written. Similarly, the programmer's experience with scripting code determines how long populating the API dictionary takes. The API dictionary for the Unity translator was populated using a graphical Library Editor. Rigorous testing of translators by multiple authors and by creating multiple games before publishing adds to development time.

Unity Translator

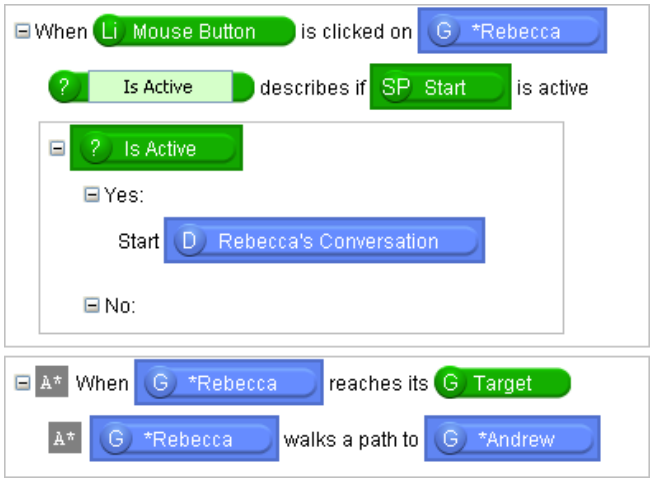

Figure 4: Two causes created with the Unity Translator.

Many events in Unity occur by checking conditions in methods that are updated at different times, such as during each frame of the game or each physics time step. If causes were built this way, there would only be a few causes and users would need to write a description and a question every time they are used. Although the translator provides this functionality, we have also included high level causes that simplify the game authoring process. For example, instead of creating a question to determine if the mouse has been clicked in the "when game frame updates" cause, the user can use a "when mouse is clicked" cause. This approach allows authors to create stories more quickly, while still supporting the flexibility of Unity's API for power users. Figure 4 shows two causes form a simple Unity story in ScriptEase. It uses a more abstract "when mouse is clicked" cause and also uses an A* pathfinding library in addition to the main Unity library.

Evaluating ScriptEase II

In 2013, SEII was used by undergraduate students in a multidisciplinary game design course (Sturtevant et al. 2008), where two thirds of the students have no programming experience. In groups of six, students spend the semester making a fully functional story-based computer game with 3D-graphics, sound and music, using the *Neverwinter Nights*

	SEI	SEII	d-value	effect
Easy to Learn	3.0	3.4	0.65	moderate
Intuitive	2.5	2.8	0.57	moderate
Easy to Organize	2.6	3.0	0.65	moderate
Easy to use Definitions/ Descriptions	2.5	2.9	0.62	moderate
Easy to use Multi-Conditions/ Questions	2.5	2.9	0.52	moderate
Easy to use Journals	2.5	3.0	0.73	moderate
Easy to use Dialogue	3.4	2.7	1.02	large

Table 1: Survey results comparing the original ScriptEase and ScriptEase II using Cohen's d-Value (1988) where moderate is $0.5 - 0.8$ and large is > 0.8 with n=10.

game engine at the University of Alberta. In the past, students were taught how to use SEI. This time, students were allowed to choose between learning SEI and SEII.

After completing tutorials on either SEI or SEII, we asked the students to complete a survey. We collected 10 completed surveys for SEI and 10 for SEII. The students were asked to rate the system they chose to use based on a number of attributes using Likert scales. Generally, each category had four possible answers to choose from, usually ranging from very difficult to very easy. We purposely left out a neutral to force an evaluation. The results were translated into a numeric value from one to four, with one corresponding to very difficult and four to very easy.

As we had only 10 results for each, we decided to look at the overall effect size instead of T-Tests (Kelley and Preacher 2012). An effect size is a descriptive statistic that conveys the estimated magnitude of a relationship. The results are shown in Table 1.

In most categories, SEII has a moderate effect size over SEI (easy to learn, intuitive, easy to organize, using descriptions, using questions, using journals). The moderate effect size for descriptions over definitions was expected, since our descriptions are based on earlier work by Desai and Szafron (2011) where descriptions were found to be easier to understand. It was also interesting that participants appeared to prefer our journal system to the SEII method, as we initially thought the new journal design might be more difficult to grasp. We also asked three questions where the effect sizes were small (d-value$<$0.5): enjoyability, ease of controlling game objects, ease of referencing game objects.

The one large effect involved dialogues, where SEI performed better than SEII. We believe the reason for this is that the interface for interacting with dialogues in SEI is modelled directly on the NWN Aurora Toolset which is sold with the official *Neverwinter Nights* game. In SEII, the dialogue lines are instead shown in the same game object pane as the rest of the game objects. This result has inspired us to construct a game-independent visual dialogue tool for SEII.

The improvements and changes in SEII have made the ScriptEase system easier to use. However, not all students in the course used ScriptEase I or II for their final game stories. The most common reason for not using ScriptEase was to access the greater expressive power that native scripting languages support. Many of the advantages of direct scripting could be obtained by customizing the translator to include new causes, effects and other story components. However, we do not yet have simple tools available for editing translators. Our next steps will include creating tools to allow users to edit the translator and examining the ability for very young authors (middle school children) to create games in SEII.

Conclusion

SEII and its story model provide a solution to an issue faced by both game studios and amateur game authors: how to proceed from designing the story to implementing it. SEII is built using high-level game patterns that support the creation of intricate stories. Non-programmers can use these patterns to easily create interactive games. SEII's game-independence means it can be adapted to any game engine by creating a translator for it. Aside from story creation, SEII is also used as a research tool for topics such as education and commercial game-related AI. It can be downloaded from http://cs.ualberta.ca/%7Escript/. Future work includes implementing behaviour patterns that support more intelligent interactions between non-player characters, story groups for further organization of the story graph, and the creation of translators for other game engines.

Acknowledgements

This research was supported by GRAND NCE, NSERC, iCORE, and Alberta Innovates Technology Futures. Thanks to the anonymous reviewers for their suggestions. Thanks to the entire ScriptEase and ScriptEase II research and development teams.

References

BioWare. 2002. Neverwinter Nights. [Video Game].

Carbonaro, M.; Szafron, D.; Cutumisu, M.; and Schaeffer, J. 2010. Computer-game construction: A gender-neutral attractor to computing science. *Computers & Education* 55(3):1098–1111.

Cohen, J. 1988. *Statistical power analysis for the behavioral sciencies*. Routledge.

Cooper, S.; Dann, W.; and Pausch, R. 2000. Alice: a 3-d tool for introductory programming concepts. In *Journal of*

Computing Sciences in Colleges, volume 15, 107–116. Consortium for Computing Sciences in Colleges.

Cutumisu, M.; Onuczko, C.; McNaughton, M.; Roy, T.; Schaeffer, J.; Schumacher, A.; Siegel, J.; Szafron, D.; Waugh, K.; Carbonaro, M.; et al. 2007. Scriptease: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming* 67(1):32–58.

Desai, N., and Szafron, D. 2011. Descriptions: a viable choice for video game authors. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, 268–270. ACM.

Fristoe, T.; Denner, J.; MacLaurin, M.; Mateas, M.; and Wardrip-Fruin, N. 2011. Say it with systems: expanding kodu's expressive power through gender-inclusive mechanics. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, 227–234. ACM.

Kelley, K., and Preacher, K. J. 2012. On effect size. *Psychological Methods* 17 (2):137–152.

MacLaurin, M. 2009. Kodu: end-user programming and design for games. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, 2. ACM.

Macvean, A.; Hajarnis, S.; Headrick, B.; Ferguson, A.; Barve, C.; Karnik, D.; and Riedl, M. O. 2011. Wequest: scalable alternate reality games through end-user content authoring. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 22. ACM.

Pizzi, D., and Cavazza, M. 2008. From debugging to authoring: Adapting productivity tools to narrative content description. In *Interactive Storytelling*. Springer. 285–296.

Resnick, M.; Maloney, J.; Monroy-Hernández, A.; Rusk, N.; Eastmond, E.; Brennan, K.; Millner, A.; Rosenbaum, E.; Silver, J.; Silverman, B.; et al. 2009. Scratch: programming for all. *Communications of the ACM* 52(11):60–67.

Skorupski, J., and Mateas, M. 2010. Novice-friendly authoring of plan-based interactive storyboards. In *AIIDE*.

Sturtevant, N. R.; Hoover, H. J.; Schaeffer, J.; Gouglas, S.; Bowling, M. H.; Southey, F.; Bouchard, M.; and Zabaneh, G. 2008. Multidisciplinary students and instructors: a second-year games course. In *ACM SIGCSE Bulletin*, volume 40, 383–387. ACM.

Trenton, M.; Szafron, D.; Friesen, J.; and Onuczko, C. 2010. Quest patterns for story-based computer games. In *Proceedings of the Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 11–13.

Unity Technologies. 2013. Unity. [Game Engine].

Unreal. 2011. Kismet visual scripting. [Video Game].

Wong, D.; Earl, D.; Zyda, F.; Zink, R.; Koenig, S.; Pan, A.; Shlosberg, S.; Singh, J.; and Sturtevant, N. 2010. Implementing games on pinball machines. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 240–247. ACM.

Zhao, R., and Szafron, D. 2009. Learning character behaviors using agent modeling in games. In *5th Annual Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-09)*, 179–185.