Generating Animations from Screenplays

Anonymous authors

Abstract

Automatically generating animation from natural language text finds application in a number of areas e.g. movie script writing, instructional videos, and public safety. However, translating natural language text into animation is a challenging task. Existing text-to-animation systems can handle only very simple sentences, which limits their applications. In this paper, we develop a text-to-animation system which is capable of handling complex sentences. We achieve this by introducing a text simplification step into the process. Building on an existing animation generation system for screenwriting, we create a robust NLP pipeline to extract information from screenplays and map them to the system’s knowledge base. We develop a set of linguistic transformation rules that simplify complex sentences. Information extracted from the simplified sentences is used to generate a rough storyboard and video depicting the text. Results show that our system performs reasonably well in terms of both efficiency and robustness.

1. Introduction

Generating animation from texts can be useful in many contexts e.g. movie script writing [Ma and Kevitt, 2006, Liu and Leung, 2006, Hanser et al., 2010], instructional videos [Lu and Zhang, 2002], and public safety [Johansson et al., 2004]. Text-to-animation systems can be particularly valuable for screenwriting by enabling faster iteration, prototyping and proof of concept for content creators. In this paper, we propose such a text-to-animation system. Given an input text describing a certain activity, the system generates a rough animation of the text. It does not aim to generate a polished, final animation, but a pre-visualization of the input text. The purpose of the system is not to replace writers and artists, but to make their work more efficient and less tedious.

Existing text-to-animation systems for screenwriting (§2) visualize stories by using a pipeline of Natural Language Processing (NLP) techniques for extracting information from texts and mapping them to appropriate action units in the animation engine. The NLP modules in these systems translate the input text into predefined intermediate action representations and the animation generation engine produces simple animation from these representations.
Although these systems can generate animation from carefully handcrafted simple sentences, translating real screenplays into coherent animation still remains a challenge. This can be attributed to the limitations of the NLP modules used with regard to handling complex sentences. In this paper, we try to address the limitations of the current text-to-animation systems. Main contributions of this paper are:

- We propose a screenplay parsing architecture which generalizes well on different screenplay formats (§3.1).
- We develop a rich set of linguistic rules to reduce complex sentences into simpler ones to facilitate information extraction (§3.2).
- We develop a new NLP pipeline to generate animation from actual screenplays (§3).
- We create a new annotated corpus for evaluating text-to-animation systems (§4 and §5.1).

In this paper, we address a more realistic setting for a text-to-animation system. In our case we do not have any annotated data for training a supervised end-to-end system. It is hoped that the techniques proposed in this paper could be used for automatically generating labelled data for text-to-animation systems.

The potential applications of our contributions are not just restricted to animating screenplays. The techniques we develop are fairly general and can be used in other applications as well. Moreover, the techniques are applicable in any setting which involves knowledge base creation. As described in §3.2, the proposed text simplification module can reduce complex and ambiguous sentences into several simple sentences. This, in combination with our information extraction module, can be used for creating knowledge bases.

2. Related Work

Translating a sentence or paragraph into animation is not a trivial task, given that neither the input sentences nor the output animations have a fixed structure. Prior work addresses this problem from different perspectives [Lu and Zhang, 2002, Hassani and Lee, 2016].

CONFUCIUS [Ma and Kevitt, 2006] is a system that converts natural language to animation using the FDG parser (a type of dependency parser) [Tapanainen and Järvinen, 1997] and WordNet [Miller, 1995]. ScriptViz [Liu and Leung, 2006] is another similar system, created for screenwriting. It uses the Apple Pie parser [Sekine, 1998] to parse input text and then recognizes objects via an object-specific reasoner. It is limited to complex sentences having conjunction between two verbs. SceneMaker [Hanser et al., 2010] adopts the same NLP techniques as proposed in CONFUCIUS [Ma and Kevitt, 2006] followed by a context reasoning module. Similar to previously proposed systems, we also use dependency parsing followed by linguistic reduction (§3.2).

Recent advances in deep learning have pushed the state of the art results on different NLP tasks [Honnibal and Johnson, 2015, Wolf et al., 2018, He et al., 2017]. We use pre-trained models for dependency parsing, coreference resolution and SRL to build a complete NLP pipeline to create intermediate action representations. For the action representation (§3.4), we use a key-value pair structure inspired by the PAR architecture [Badler et al., 2000], which is a knowledge base of representations for actions performed by virtual human agents.
Figure 1: System Architecture: Screenplays are first segmented into different functional blocks. Then, the descriptive action sentences are simplified. Simplified sentences are used to generate animation.

Our work comes close to the work done in the area of Open Information Extraction (IE) [Niklaus et al., 2018]. In particular, to extract information, Clause-Based Open IE systems [Del Corro and Gemulla, 2013, Angeli et al., 2015, Schmidek and Barbosa, 2014] reduce a complex sentence into simpler sentences using linguistic patterns. However, the techniques developed for these systems do not generalize well to screenplay texts, as these systems have been developed using well-formed and factual texts like Wikipedia, Penn TreeBank, etc. An initial investigation with the popular Open IE system OLLIE (Open Language Learning for Information Extraction) [Mausam et al., 2012] did not yield good results on our corpus.

Previous work related to information extraction for narrative technologies includes the CARDINAL system [Marti et al., 2018, Sanghrajka et al., 2018], as well as the conversational agent PICA [Falk et al., 2018]. They focus on querying knowledge from stories. The CARDINAL system also generates animations from input texts. However, neither of the tools can handle complex sentences. Our system can be seen as an advanced version of the CARDINAL system, where we develop a new NLP module to support complex sentences and leverage the animation engine of CARDINAL.

3. Text-to-Animation System

We adopt a modular approach for generating animations from screenplays. The general overview of our approach is presented in Figure 1. The system is divided into three modules:

- **Script Parsing Module**: Given an input screenplay text, this module automatically extracts the relevant text for generating the animation (§3.1).
- **NLP Module**: It processes the extracted text to glean relevant information. This has two sub-modules:
  - **Text Simplification Module**: It simplifies complex sentences using a set of linguistic rules (§3.2).
  - **Information Extraction Module**: It extracts information from the simplified sentences into a pre-defined action representations (§3.4).
• **Animation Generation Module:** It generates animation based on action representations (§3.5).

### 3.1 Script Parsing Module

Typically, *screenplays* or *movie scripts* or *scripts* (we use the terms interchangeably), are made of several scenes, each of which corresponds to a series of consecutive motion shots. Each scene contains several functional components\(^1\): *Heading* (time and location), *Descriptions* (scene description, character actions), *Character Cues* (character name before dialog), *Dialogs* (conversation content), *Slug Lines* (actions inserted into continuous dialog) and *Transitions* (camera movement). In many scripts, these components are easily identifiable by indentation, capitalization and keywords. We call these scripts *well-formatted*, and the remaining ones as *ill-formatted*. We want to segment the screenplays into components and are mainly interested in the *Description* component for animation generation.

**Well-formatted Scripts:** We initially tried ScreenPy [Winer and Young, 2017] to annotate the well-formatted scripts with the component labels. However, ScreenPy being a grammar based tool, follows a very rigid grammar and fails to annotate noisy scripts. We developed our own model for parsing scripts. Since well-formatted screenplays are indented in a consistent way, we propose using a Finite State Machine (FSM) model for extracting all components (shown in Figure 2) with seven state transition rules as summarized in Table 1. Each component corresponds to a state in the FSM. The screenplays are fed to the FSM line by line. The process always starts from the *Heading* state. Depending on the indentation difference between the current and the previous line, the FSM either saves the current line to current component or transitions to another state. In the end, the model segments the input screenplay and generates (paragraph, component name) pairs.

**Ill-formatted Scripts:** Taking all the (paragraph, component name) pairs generated by the FSM as ground truth, an SVM model is trained to segment ill-formatted screenplays with inconsistent indentations. For extracting features, each paragraph is encoded into a

---

1. https://www.storysense.com/format.htm
fix sized vector using a Universal Sentence Encoder\(^2\) [Cer et al., 2018]. The SVM is trained and tested on a 9:1 train-test data split. The result shows an accuracy of 92.72% on the test set, which is good for our purposes as we are interested mainly in the Description component.

**Coreference Resolution:** Natural language is sometimes ambiguous due to different mentions for the same entity, particularly with pronouns. This phenomenon is commonly seen in screenplays. In order to link mentions of an entity, an accurate coreference resolution system is required.

After labeling paragraphs of the script with their component labels, we select the Description components, since these describe the actions in the screenplay. Each of the paragraphs is processed with the NeuralCoref\(^3\). Given the text, it resolves mentions (typically pronouns) to the entity they refer to in the text.

Unlike most text corpora which are coherent in narrative, screenplays are composed of different functional parts and thus less coherent. To get correct entity names in each of the Description blocks, we prepend it with the Character Cues component which appears before it in the screenplay (e.g. [character]MARTHA: [dialog]“I knew it!” [description]She then jumps triumphantly. She then jumps triumphantly). This concatenated text goes as the input to the NeuralCoref model.

### 3.2 Text Simplification Module

In a typical text-to-animation system, one of the main tasks is to process the input text to extract the relevant information about actions (typically verbs) and participants (typically syntactic subject/object of the verb), which is subsequently used for generating animation. This works well for simple sentences having a single verb with one subject and one (optional) object. However, the sentences in a screenplay are complicated and sometimes informal. In this work, a sentence is said to be complicated if it deviates from easily extractable and simple subject-verb-object (and its permutations) syntactic structures (§3.2.5 to §3.2.10) and possibly has multiple actions mentioned within the same sentence with syntactic interactions between them. By syntactic structure we refer to the dependency graph of the sentence.

In the case of screenplays, the challenge is to process such complicated texts. We take the text simplification approach, i.e. in our system, we first simplify the complicated sentence and then extract the relevant information. Simplification reduces a complicated sentence into multiple simpler sentences, each having a single action along with its participants, making it straightforward to extract necessary information.

Recently, end-to-end Neural Text Simplification (NTS) systems [Nisioi et al., 2017, Saggion, 2017] have shown to achieve reasonable accuracy. However, these systems have been trained on factual data such as Wikipedia and do not generalize well to screenplay texts. Our experiments with such a pre-trained neural text simplification system yielded very poor results on our data. Moreover, in the context of text-to-animation, there is no standard labeled corpus to train an end-to-end system. Based on the experiments with the screenplay texts, we develop linguistic rules to simplify these texts. However, the rules

\(^2\) We use the pre-trained implementation [https://tfhub.dev/google/universal-sentence-encoder/2](https://tfhub.dev/google/universal-sentence-encoder/2)

\(^3\) [https://github.com/huggingface/neuralcoref](https://github.com/huggingface/neuralcoref)
developed here are fairly general and can also be applied for extracting information from other types of texts for populating knowledge bases.

Algorithm 1: Syntactic Simplification Procedure

```python
1: procedure Syntactic Simplification(sent, temp)
2: Q ← empty queue
3: HS ← empty integer hash set
4: RES ← empty list
5: Q.push(sent)  # push input sentence to queue
6: while Q ≠ Empty do
7:     str ← Q.pop()
8:     if hash(str) ∈ HS then
9:         RES.append(str)  # have seen this sentence
10:     continue
11:     HS.add(str)  # mark current sentence as already seen
12:     transform ← False
13:     for a in analyzers do
14:         if !transform & a.identify(str) then
15:             transform ← True
16:             simplified = a.transform(str)
17:             correct_verb_tense(simplified)
18:             Q.push(simplified)
19:         if transform ≠ True then
20:             RES.append(str)
21:     return RES
```

There has been work on text simplification using rule-based approaches. For example, [Siddharthan, 2011] propose a set of rules to manipulate sentence structure to output simplified sentences using syntactic dependency parsing. Similarly, the YATS system [Ferrés et al., 2016] implements a set of rules in the JAPE language [Cunningham et al., 2000] to address six syntactic structures: Passive Constructions, Appositive Phrases, Relative Clauses, Coordinated Clauses, Correlated Clauses and Adverbial Clauses. Most of the rules focus on case and tense correction, with only 1-2 rules for sentence splitting. The coverage of their work is limited. We take inspiration from the YATS system, and our system incorporates modules to identify and transform sentence structures into simpler ones using a larger set of rules.

In our system, each syntactic structure is handled by an Analyzer, which contains two processes: Identify and Transform. The Identify process takes in a sentence and determines if it contains a particular syntactic structure. The Transform process takes the same sentence as input, focuses on the first occurrence of the identified syntactic structure and, then splits and assembles the sentence into 1 or 2 simpler sentences. Both Identify and Transform use Part of Speech (POS) tagging and dependency parsing [Honnibal and Montani, 2017] modules implemented in SpaCy 2.0⁴ [Honnibal and Montani, 2017].

The simplification algorithm (Algorithm 1) starts with an input sentence and recursively processes it until no further simplification is possible. It uses a queue to manage intermediate simplified sentences, and runs in a loop until the queue is empty. For each sentence, the system applies each syntactic analyzer to Identify the corresponding syntactic structure in the sentence (line 14). If the result is positive, the sentence is processed by the Transform function to convert it to simple sentences (line 16). Each of the output sentences is pushed by the controller into the queue (line 19). The process is repeated with each of the Identify

⁴. https://spacy.io
Table 2: Syntactic simplification rules applied on example sentences.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Input Sentence</th>
<th>System Output Sentence 1</th>
<th>System Output Sentence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination</td>
<td>She LAUGHS, and [cc] gives [conj] Kevin a kiss.</td>
<td>She laughs.</td>
<td>She gives Kevin a kiss.</td>
</tr>
<tr>
<td>Pre-Correlative</td>
<td>It’s followed by another squad car, both [preconj] with sirens blaring.</td>
<td>It’s followed by another squad car, with sirens blaring.</td>
<td>–</td>
</tr>
<tr>
<td>Appositive</td>
<td>Kevin is reading a book the Bible [appos]</td>
<td>Kevin reads a book</td>
<td>The book is the Bible.</td>
</tr>
<tr>
<td>Relative-dobj</td>
<td>She pulls out a letter which [dobj] she hands [relcl] to Kevin.</td>
<td>She pulls out a letter</td>
<td>She hands a letter to Kevin.</td>
</tr>
<tr>
<td>Relative-pobj</td>
<td>A reef encloses the cove where [pobj] he came [relcl] from.</td>
<td>A reef encloses the cove</td>
<td>he comes from the cove.</td>
</tr>
<tr>
<td>Relative-nsubj</td>
<td>Frank gestures to the SALESMAN, who [nsubj]'s waiting [relcl] on a woman.</td>
<td>the SALESMAN waits on a woman.</td>
<td>Frank gestures to the SALESMAN.</td>
</tr>
<tr>
<td>Relative-advmod</td>
<td>Chuck is in the stage of exposure where [advmod] the personality splits [relcl].</td>
<td>Chuck is in the stage of exposure</td>
<td>the personality splits at exposure.</td>
</tr>
<tr>
<td>Relative-poss</td>
<td>The girl, whose [poss] name is [relcl] Helga, cows.</td>
<td>The girl cows</td>
<td>The girl’s name is Helga</td>
</tr>
<tr>
<td>Relative-omit</td>
<td>Kim is the sexpot Peter saw [relcl] in Washington Square Park</td>
<td>Peter sees Kim in Washington Square Park.</td>
<td>Kim is the sexpot.</td>
</tr>
<tr>
<td>Adverbial- remove</td>
<td>Suddenly there’s a KNOCK at the door, immediately after [prep] which JIM’S MOM enters [advcl].</td>
<td>Suddenly there’s a KNOCK at the door.</td>
<td>Immediately JIM’S MOM enters.</td>
</tr>
<tr>
<td>Inverted Cl. Subject</td>
<td>Running [subj] towards Oz is Steve Stifler.</td>
<td>Steve Stifler runs towards Oz.</td>
<td>–</td>
</tr>
<tr>
<td>Passive Voice</td>
<td>They [nsubjpass] are suddenly illuminated by the glare of headlights.</td>
<td>Suddenly the glare of headlights illuminated them.</td>
<td>It actually sounds really good.</td>
</tr>
<tr>
<td>Open Clausal</td>
<td>The sophomore comes running [xcomp] through the kitchen.</td>
<td>The sophomore runs through the kitchen.</td>
<td>The sophomore comes.</td>
</tr>
<tr>
<td>Adjective</td>
<td>Stifler has a toothbrush hanging [acl] from his mouth.</td>
<td>A toothbrush hangs from Stifler’s mouth.</td>
<td>Stifler has a toothbrush.</td>
</tr>
</tbody>
</table>

3.2.1 Coordination

Coordination is used for entities having the same syntactic relation with the head and serving the same functional role (e.g. subj, obj, etc.). It is the most important component in our simplification system. The parser tags word units such as ‘and’ and ‘as well as’ with the dependency label cc, and the conjugated words as conj. Our system deals with coordination based on the dependency tag of the conjugated word.

In the case of coordination, Identify function simply returns whether cc or conj is in the dependency graph of the input sentence. The Transform function manipulates the graph structure based on the dependency tags of the conjugated words. If the conjugated word is a verb, then we mark it as another root of the sentence. Cutting cc and conj edges in the graph and traversing from this new root results in a new sentence parallel to the original one. In other cases, such as the conjugation between nouns, we simply replace...
the noun phrases with their siblings and traverse from root again (Algorithm 6 in §A). In coordination-structured sentences, sometimes the POS tagger tends to tag verbs as nouns. We employ heuristic rules to correct such mistagging mistakes.

3.2.2 PRE-CORRELATIVE CONJUNCTION

A Pre-Correlative Conjunction (preconjunct) is the relation between the head of an NP and a word that is part of a conjunction, and puts emphasis on it (e.g., “either”, “both”, “neither”) [De Marneffe and Manning, 2008]. The Identify function in the preconjunct analyzer locates the position of keywords: “either”, “both”, “neither”. The Transform function removes the word. For Neither, we replace it with the word Not. This step may sometimes distort the original meaning. In the future, we plan to address this problem.

3.2.3 APPOSITIVE CLAUSE

Appositive Clauses are usually NPs that are used for emphasis or further explanation of another NP. In Table 2, the Bible is the apposition anchor of book. The Identify method in the apposition analyzer loops over all tokens in the input sentence. It returns True only if it finds a token with the dependency tag appos, and the token’s head is a noun. Otherwise it returns False. The Transform method (Algorithm 9 in §A) glues the subject part with the appositive noun part using a suitable form of the verb “to be” to get the sentence.

3.2.4 RELATIVE CLAUSE

A relative clause is a subordinate of the antecedent clause on which it depends. Sometimes they have a function similar to that of appositive clauses. The dependency tag for a relative clause anchor is relcl. The Identify function finds the corresponding anchor token and its head token (Algorithm 4 in §A). There are two cases to consider: with or without a ‘wh-’ word in the sentence (examples in Table 2). First output sentence is generated by cutting the recl edge in the dependency graph and then in-order traversal from the root token. For generating the second sentence, we need to consider different cases. For sentences without an ‘wh’ word, we simply put the head part after the anchor part. For sentences with ‘wh’
words, depending on the different possible functionality of each ‘wh’ word, we divide them into five sub-cases (Algorithm 5 in §A), also shown in Table 2.

3.2.5 Adverbial Clause Modifier

An adverbial clause modifier of a verb phrase (VP) is a clause modifying the verb (temporal clause, consequence, conditional clause, etc.) [De Marneffe and Manning, 2008]. These are frequently used in screenplay writing. The dependency between an adverbial verb and the main verb is marked as *advcl*. We list two examples in Table 2. The *Identify* and *Transform* functions split the sentence into two with the subject copied to both. For more details refer to Algorithm 2 and Algorithm 3 in §A.

3.2.6 Inverted Clausal Subject

If a clause functions as a subject in a sentence, we do not modify it, because in most cases it describes something that cannot be animated, such as in the sentence *Where the rock had chipped the edge is sharp*. But in other cases where a clausal subject appears in an inverted sentence i.e. the verb coming before the subject, we need to modify the sentence structure to get the correct action meaning. Table 2 shows one example. We use the *csubj* tag to identify clausal subjects. We identify the inverted sentence with the help of the *attr* tag for the node. The node which is tagged as *attr* must be present in the child node of the head of the *csubj* node (Algorithm 10 in §A). After identifying the structure, the *transform* method simply changes the position of the actual verb and the actual subject (Algorithm 11 in §A).

3.2.7 Clausal Complement

A Clausal Complement is used as the modifier to describe the previously mentioned event in detail. It is tagged as *ccomp* in the dependency graph. In screenplays, most of the cases of clausal complements cannot be animated, with some exceptions. We use the *ccomp* tag to identify clausal complements in a sentence. We then cut the *ccomp* edge and add a subject to the subordinate clause if necessary (Algorithm 12 in §A). However, in some cases, the output sentences are not naturally complete. For the example sentence shown in Table 2, the first output sentence is *The thing is*, which is meaningless to animation generation module. Hence, we place the clausal complement analyzer near the end of the analyzer’s list to avoid potential mistakes.

3.2.8 Passive Voice

Converting passive voice to active voice is crucial for getting correct output from the system. Passive voice sentence contains the dependency labels: *nsubjpass*, *csubjpass* (the passive subject). Sometimes, dependency labels *auxpass* (verb “to be”) and *agent* (indicates the entity that actually conducts this action) are also present. In the *Identify* function, we check the presence of *nsubjpass* or *csubjpass* labels, and optionally for the other two labels (Algorithm 7 in §A). The *Transform* (Algorithm 8 in §A) procedure reorders the dependency graph and corrects the verb tense to express the same sentence in active voice.
3.2.9 Open Clause Complement

An Open Clausal Complement is usually the predicate of a verb or an adjective. In the dependency parser, it has the dependency label \textit{xcomp}. The \textit{Identify} function simply finds the main verb and the auxiliary verb. The \textit{Transform} function substitutes the auxiliary verb with the main verb in the dependency graph and traverses it to get the output sentence. In some cases, when we find the \textit{aux} tag and we remove it, then traverse from both verbs (Algorithm 13 in §A). An example sentence is shown in Table 2.

3.2.10 Adjective Clause

An Adjective Clause is used to provide descriptive information for an object in a screenplay. It is tagged as \textit{acl} by the dependency parser. The \textit{Identify} function simply marks the acl-verb-token and its head token. The \textit{Transform} function first cuts the acl edge and traverses the root token to output the first sentence. Then it examines if the adjective modifier contains any special cases (Algorithm 14 in §A).

3.3 Lexical Simplification

In order to generate animation, actions and participants extracted from simplified sentences are mapped to existing actions and objects in the animation engine. Due to practical reasons, it is not possible to create a library of animations for all possible actions in the world. We limit our library to a predefined list of 52 actions/animations, expanded to 92 by a dictionary of synonyms (§3.5). We also have a small library of pre-uploaded objects (such as “campfire”, “truck” and others). Nevertheless, this is a work in progress and we are working on including more animations for actions and objects in our knowledge base.

To animate unseen actions not in our list, we use a word2vec-based similarity function to find the nearest action in the list. Moreover, we use WordNet [Miller, 1995] to exclude antonyms. This helps to map non-list actions (such as “squint at”) to the similar action in the list (e.g. “look”). If we fail to find a match, we check for a mapping while including the verb’s preposition or syntactic object. We also use WordNet to obtain hypernyms for further checks, in the case similarity function fails to find a close-enough animation. Correspondingly, for objects, we use the same similarity function and WordNet’s holonyms. In future work, we plan to leverage more functionality from WordNet, such as meronyms or hyponyms and we also plan train word embeddings on screenplay texts in order to generalize better.

3.4 Action Representation Field (ARF): Information Extraction

For each of the simplified sentences, we extract information from it and populate it in a predefined key-value pair structure. We will refer to the keys of this structure as Action Representation Fields (ARFs). These are similar to entities and relations in Knowledge Bases. ARFs include: owner, target, prop, action, origin_action, manner, modifier_location, modifier_direction, start-time, duration, speed, translation, rotation, emotion, partial_start_time (§B). This structure is inspired by the PAR [Badler et al., 2000] architecture, but adapted to our needs.

The \textit{owner} roughly corresponds to the sentence’s syntactic subject, the \textit{action} to the verb, the \textit{target} and \textit{prop} to syntactic objects (both direct and indirect). The \textit{duration} field
Generating Animations from Screenplays

Figure 4: Pre-visualization interface

is modified if the sentence contains words such as “slowly” or “quickly”, which indicate that the animation should last for a longer or shorter time. When words like “angrily” or “lazily” are used, then speed (the quickness of the animation movement) is affected. translation indicates if an action entails movement from one point to another (e.g. “go”) and rotation indicates movement in place (e.g. “turn”, “sit”). partial_start_time contains the temporal order of the simplified sentences.

To extract the ARFs from the simplified sentences, we use a Semantic Role Labelling model (SRL) model in combination with some heuristic rules, such as finding matches in word lists for duration, speed, translation, rotation, emotion. We use a pre-trained Semantic Role Labelling model based on a Bi-directional LSTM network [He et al., 2017] with pre-trained ELMo embedding [Peters et al., 2018]. We map information from each sentence to the knowledge base of animations and objects.

3.5 Animation Generation

We use the animation pipeline of the CARDINAL system. We plug in our NLP module in CARDINAL to generate animation. CARDINAL creates pre-visualizations of the text, both in storyboard form and animation. A storyboard is a series of pictures that demonstrates the sequence of scenes from a script. The animation is a 3-D animated video that approximately depicts the script. CARDINAL uses the Unreal game engine [Games, 2007] for generating pre-visualizations. It has a knowledge base of pre-baked animations (52 animations such as “go”, “laugh”, plus a dictionary of synonyms, resulting in 92) and pre-uploaded objects (e.g. campfire, tent). It also has 3-D models which can be used to create characters. The ARFs extracted with the NLP module (§3.4) are used to generate the pre-visualizations (Figure 4). The ARF owner usually refers to a character; the target and prop are the characters or objects and the ARF action is an action in the knowledge base. Other ARFs, such as speed or duration also inform the pre-visualization.

5. the model is available in the AllenNLP library: https://github.com/allenai/allennlp
4. Text-to-Animation Corpus

We initially used a corpus of Description components from ScreenPy [Winer and Young, 2017], in order to observe linguistic patterns in the movie script domain. Specifically, we used the “heading” and “transition” fields from ScreenPy’s published JSON output on 1068 movie scripts scraped from IMSDb. We also scraped screenplays from SimplyScripts\(^6\) and ScriptORama\(^7\). After separating screenplays into well-formatted and ill-formatted, Description components were extracted using our model (§3.1). This gave a corpus of Description blocks from 996 screenplays.

The corpus contains a total of 525,708 Description components. The Description components contain a total of 1,402,864 sentences. Out of all the Description components, 49.45% (259,973) of these components contain at least one verb which is in the animation list (henceforth called “action verbs”). Description components having at least one action verb have in total 920,817 sentences. Out of these, 42.2% (388,597) of the sentences contain action verbs. In the corpus, the average length of a sentence is 12 words. We split the corpus into training (796 scripts), development (100 scripts) and test sets (100 scripts).

5. Evaluation and Analysis

There are no standard corpora for text-to-animation generation. It is also not clear how should such systems be evaluated and what should be the most appropriate evaluation metric. Nevertheless, it is important to assess how our system is performing. We evaluate our system using two types of evaluation: Intrinsic and Extrinsic. Intrinsic evaluation is for evaluating the NLP pipeline of our system using the BLEU metric. Extrinsic evaluation is an end-to-end qualitative evaluation of our text-to-animation generation system, done via a user study.

5.1 Intrinsic Evaluation

To evaluate the performance of our proposed NLP pipeline, 500 Description components from the test set were randomly selected. Four annotators manually translated these 500 Description components into simplified sentences and extracted all the necessary ARFs from the simplified sentences. This is a time intensive process and took around two months. 30% of the Description blocks contain verbs not in the list of 92 animation verbs. There are approximately 1000 sentences in the test set, with average length of 12 words. Each Description component is also annotated by the four annotators for the ARFs. Upon acceptance, we plan to release this gold dataset. The dataset would serve as a benchmark for evaluating text-to-animation systems.

Taking inspiration from the text simplification community [Nisioi et al., 2017, Saggion, 2017], we use the BLEU score [Papineni et al., 2002] for evaluating our simplification and information extraction modules. For a simplified sentence \(s\), we have 4 corresponding references \(r_1, r_2, r_3, r_4\). We also evaluate using the SARI [Xu et al., 2016] score to evaluate our text simplification module. Recently, [Sulem et al., 2018] have empirically shown that BLUE is not a perfect metric for evaluating text simplification systems.

\(^6\) http://www.simplyscripts.com
\(^7\) http://www.script-o-rama.com
Table 3: Results on syntactic simplification

<table>
<thead>
<tr>
<th>System</th>
<th>BLEU</th>
<th>SARI</th>
</tr>
</thead>
<tbody>
<tr>
<td>YATS</td>
<td>69.17</td>
<td>36.28</td>
</tr>
<tr>
<td>Our System</td>
<td>75.44</td>
<td>37.58</td>
</tr>
</tbody>
</table>

Table 4: Differences between system output and annotator responses

<table>
<thead>
<tr>
<th>System output</th>
<th>Annotator I</th>
<th>Annotator II</th>
<th>BLEU2(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carl touches Ellie’s shoulder as the doctor explains. Ellie drops her head in her hands.</td>
<td>carl touches elle’s shoulder</td>
<td>carl touches elle’s shoulder</td>
<td>38.73</td>
</tr>
<tr>
<td>the doctor explains</td>
<td>the doctor explains</td>
<td>the doctor is talking.</td>
<td>100</td>
</tr>
<tr>
<td>Ellie drops Ellie head in Ellie hands</td>
<td>ellie drops her head in her hands</td>
<td>ellie drops her head in her hands</td>
<td>48.79</td>
</tr>
</tbody>
</table>

5.1.1 Sentence Simplification

We measure using BLEU and SARI scores at sentence level to evaluate the performance of our simplification module. The results are summarized in Table 3. YATS [Ferrés et al., 2016] is also a rule based text simplification system. As shown in Table 3, our system performs better than YATS on both the metrics, indicative of the limitations of the YATS system. A manual examination of the results also showed the same trend (Table 8 in Appendix A). However, the key point to note is that we are not aiming for text simplification in the conventional sense. Existing text simplification systems tend to summarize text and discard some of the information. Our aim is to retain the information and break a complex sentence into simpler ones having the information.

An example of a Description component with BLEU2 scores is given in Table 4. In the first simplified sentence, the space between Ellie and ’s causes the drop in the score. But it gives exactly the same answer as both annotators. In the second sentence, the system output is the same as the annotator I’s answer, so the BLEU2 score is 1. In the last case, the score is low, as annotators possibly failed to replace her with the actual Character Cue Ellie. In general, our system gives a reasonable result for the syntactic simplification module. As exemplified, BLEU is not the perfect metric to evaluate our system, and therefore in the future we plan to explore other metrics.

5.1.2 ARF Evaluation

We also evaluate the system’s output for action representation fields against gold annotations. In our case, some of the fields can have multiple (2 or 3) words such as owner, target, prop, action, origin_action, manner, location and direction. We use BLEU1 as the evaluation metric to measure the BOW similarity between system output and ground truth references. The results are shown in Table 5.

In identifying owner, target and prop, the system tends to use a fixed long mention, while annotators prefer short mentions for the same character/object. The score of prop is relatively lower than all other fields, which is caused by a systematic SRL mapping error. The relatively high accuracy on the action field indicates the consistency between system output and annotator answers. The NLP pipeline gives reasonable results on other ARFs.
Annotation on the emotion ARF is rather subjective. Responses on the this field are biased and noisy. The BLEU$_1$ score on this is relatively low. For the other non-textual ARFs, we use precision and recall to measure the system’s behavior. Results are shown in Table 6. These fields are subjective: annotators tend to give different responses for the same input sentence.

rotation and translation have Boolean values. Annotators agree on these two fields in most of the sentences. The system, on the other hand, fails to identify actions involving rotation. For example, in the sentence “Carl closes CARL’s door sharply” all four annotators think that this sentence involves rotation, which is not found by the system. This is due to the specificity of rules on identifying these two fields.

speed, duration and start_time have high precision and low recall. This indicates the inconsistency in annotator’s answers. For example, in the sentence “Woody runs around to the back of the pizza truck”, two annotators give 2 seconds and another gives 1 second in duration. These fields are subjective and need the opinion of the script author or the director. In the future, we plan to involve script editors in the evaluation process.

### 5.2 Extrinsic Evaluation

We conducted a user study to evaluate the performance of the system qualitatively. The focus of the study was to evaluate (from the end user’s perspective) the performance of the NLP component w.r.t. generating reasonable animations.

We developed a questionnaire consisting of 20 sentence-animation video pairs, the animations were generated by our system. The questionnaire was filled by 22 participants. On an average it took around 25 minutes for a user to complete the study.

We asked users to evaluate, on a five-point Likert scale [Likert, 1932], if the video shown was a reasonable animation for the text, how much of the text information was depicted in the video and how much of the information in the video was present in the text (Table 7).

The 68.18% of the participants rated the overall pre-visualization as neutral or above. The rating was 64.32% (neutral or above) for the conservation of textual information in the
video, which is reasonable, given limitations of the system that are not related to the NLP component. For the last question, 75.90% (neutral or above) agreed that the video did not have extra information. In general, there seemed to be reasonable consensus in the responses. Besides the limitations of our system, disagreement can be attributed to the ambiguity and subjectivity of the task.

We also asked the participants to describe qualitatively what textual information, if any, was missing from the videos. Most of the missing information was due to limitations of the overall system rather than the NLP component: facial expression information was not depicted because the character 3-D models are deliberately designed without faces, so that animators can draw on them. Information was also missing in the videos if it referred to objects or actions that do not have a close enough match in the object list or animations list. Furthermore, the animation component only supports animations referring to a character or object as a whole, not parts, (e.g. “Ben raises his head” is not supported).

However, there were some cases where the NLP component can be improved. For example, lexical simplification failed to map the verb “watches” to the similar animation “look”. In one case, syntactic simplification created only two simplified sentences for a verb which had three subjects in the original sentence. In a few cases, lexical simplification successfully mapped to the most similar animation (e.g.“argue” to “talk”) but the participants were not satisfied - they were expecting a more exact animation. We plan to address these shortcomings in future work.

6. Conclusion and Future Work

In this paper, we proposed a new text-to-animation system. We also introduced new corpora for evaluating the system. Our system gives promising results, particularly the script segmentation module and syntactic simplification module. However, it has limits on how much it can generalize, due to the heuristic, rule-based nature of our techniques, as well as our dependence on a finite list of pre-baked animations. The system performance is also dependent on the accuracy of the external tools in the pipeline, such as syntactic parsing and coreference resolution. Evaluating such systems is a challenge. Nevertheless, we evaluated our system using intrinsic and extrinsic evaluation. The evaluation shows reasonable performance of the system.

The NLP module in our system is not perfect and we plan to improve it in the future based on evaluation and user feedback. Possibly, in the future we plan to use the simplified sentences we obtained from the system to train an end-to-end sentence simplification system using deep learning techniques [Nisioi et al., 2017, Saggion, 2017]. The current system does not take into account the discourse information that links the actions implied in the text, as currently the system only processes sentences independently. In the future, we would like to leverage discourse information by considering the sequence of actions which are described in the text. There has been work in the area of modeling event chains and narrative scripts [Modi and Titov, 2014, Modi, 2016] and we would like explore this line of research. As shown in the paper, evaluation metrics used are not perfect, and we plan to explore more in this direction.
Anonymous authors

References


Generating Animations from Screenplays


Appendix A: Algorithms

Algorithm 2 Identify Adverbial Clause

1: procedure Adverbial Clause Identify Procedure(sent) \( \triangleright \) The input sentence  
2: find tokens in sent with dependency tag ROOT and ADVCL (we call it advcl)  
3: if no ADVCL token in sent then  
4: return False  
5: find father token of advcl as father  
6: if father is not a VERB then  
7: we make it as a VERB \( \triangleright \) We correct Spacy error here  
8: find conjunction tokens of father as conjuncts  
9: if NOUN in advcl’s left subtree then  
10: subject \( \leftarrow \) this NOUN  
11: else  
12: subject \( \leftarrow \) NOUN in father’s left subtree  
13: return True

Algorithm 3 Transform Adverbial Clause

1: procedure Adverbial Clause Transform Procedure(sent) \( \triangleright \) The input sentence  
2: cut edge between root and advcl token \( \triangleright \) remove advcl token from root’s children  
3: if advcl verb does not have it’s own subject then  
4: add subject as advcl’s most left direct child  
5: remove PREP and MARK token in advcl’s children, modify temporal id accordingly  
6: str1 \( \leftarrow \) traverse_a_string(root)  
7: str2 \( \leftarrow \) correct_tense(traverse_a_string(advcl))  
8: return [str1, str2]

Algorithm 4 Identify() in Relative Clause Analyzer

1: procedure Relative Clause Identify Procedure(sent) \( \triangleright \) The input sentence.  
2: if no RELCL token in sent then  
3: return False  
4: anchor \( \leftarrow \) RELCL token  
5: head \( \leftarrow \) anchor’s head token  
6: wg \( \leftarrow \) NULL  
7: for token t in anchor’s children do  
8: if t.tag \( \in \) [WDT, WP, WP$, WRB] then  
9: wh \( \leftarrow \) t  
10: return True
Algorithm 5 Transform() in Relative Clause Analyzer

1: procedure RelativeClauseTransform Procedure(root, anchor, head, wh)  \(\triangleright\) Root of dependency tree, the repl anchor token, its head, and wh-word
2:     cut relcl edge between head and anchor
3:     str1 ← traverse_a_string(root)
4:     if wh = NULL then  \(\triangleright\) No Wh-word in the sentence, concatenate
5:         str2 ← traverse_a_string(anchor) + traverse_a_string(head)
6:     else
7:         wh-dep ← dependency tag of wh
8:         wh-head ← head of wh
9:         remove wh in anchor’s children
10:         if wh-dep= DOBJ then  \(\triangleright\) wh is verb
11:             str2 ← traverse_a_string(wh-head) + traverse_a_string(anchor)
12:         else if wh-dep= POBJ then  \(\triangleright\) wh-head is preposition
13:             put head after wh-head in anchor’s children
14:             str2 ← traverse_a_string(anchor)
15:         else if wh-dep ∈ [NSUBJ, NSUBJPASS] then  \(\triangleright\) wh is subject
16:             str2 ← traverse_a_string(wh-head) + traverse_a_string(anchor)
17:         else if wh-dep= ADVMOD then  \(\triangleright\) wh is time or location
18:             prep ← ‘at’
19:             str2 ← traverse_a_string(anchor) + prep + traverse_a_string(wh-head)
20:         else if wh-dep= POSS then  \(\triangleright\) wh = whose
21:             str2 ← traverse_a_string(wh-head) + be verb + traverse_a_string(anchor)
22:         correct verb tense in str1 and str2
23:     return [str1, str2]

Algorithm 6 Transform() in Coordination Analyzer

1: procedure CoordinationTransform Procedure(sents)  \(\triangleright\) Input sentence
2:     results ← empty list
3:     find root of dependency tree of sents
4:     find first cc (if any) and conj token in dependency tree and their head token main
5:     embed all conjugate words of main in a list conjus
6:     cut conj edge between main and cc and conj
7:     if no object for main then
8:         try find object in conj’s right children
9:     str1 ← traverse_a_string(root)
10:    results.append(str1)
11:    for conj in conjus do
12:        type ← get_conj_type(main, conj)
13:        if type=VERB&VERB then  \(\triangleright\) Spacy tends to tag verb as noun
14:            correct part-of-speech tag if necessary
15:        if conj has its own subject then
16:            new-root ← conj
17:        else
18:            if main=root then
19:                new-root ← conj
20:        else
21:            replace main with conj in root’s children
22:            new-root ← root
23:        else  \(\triangleright\) Other cases such as NOUN&NOUN, AD*&AD*, apply same rule
24:            main-head ← head of main
25:            replace main with conj in main-head’s children
26:            new-root ← root
27:        str2 ← traverse_a_string(new-root)
28:    results.append(str2)
29:    return results
Algorithm 7 Identify() in Passive Analyzer
1: procedure PASSIVE_Voice IDENTIFY Procedure(sents) ▷ Input sentence
2: is-passive ← False
3: for token t in sents do
4: if t.dep ∈ [CSUBJPASS, NSUBJPASS] then
5:   subj-token ← t
6:   verb-token ← t.head
7:   is-passive ← True
8: if t.dep ∈ [AUXPASS] and t.head = verb-token then
9:   auxpass-token ← t
10: if t.dep ∈ [AGENT] and t.text = 'by' then
11:   by-token ← t
12: return is-passive

Algorithm 8 Transform() in Passive Analyzer
1: procedure PASSIVE_Voice TRANSFORM Procedure(sents) ▷ Input sentence
2: if auxpass-token ̸= NULL then
3:   cut auxpass edge
4:   cut nsubjpass or csubj edge
5:   prepend subject-token to verb-token’s right children
6: if by-token ̸= NULL then
7:   cut by-agent edge
8:   add by-token’s right children to verb-token’s left children
9: else
10:   add ‘Sombody’ to verb-token’s left children
11: correct verb tense for verb-token
12: return traverse_a_string(root-token)

Algorithm 9 Transform() in Appositive Clause Analyzer
1: procedure APPOSITIVE_CLAUSE TRANSFORM Procedure(anchor, head) ▷ The APPOS token and it’s head token.
2: cut edge between anchor and head token ▷ remove anchor from head’s right children
3: str1 ← traverse_a_string(root token of input sentence)
4: str2 ← traverse_a_string(head) + be_verb + traverse_a_string(anchor)
5: correct verb tense in str1 and str2
6: return [str1, str2]

Algorithm 10 Identify() in Inverted Clausal Subject Analyzer
1: procedure INVERTED_CLAUSAL_SUBJECT IDENTIFY Procedure(sents) ▷ Input sentence
2: for Token t in sents do
3:   if t.dep = CSUBJ and t.tag ∈ [VBN, VBG] and t.head.lemma='be' then
4:     attr ← token with dependency label attr in t.head’s right children
5:     if attr = NULL then ▷ attr is the actual subject of the sentence
6:       return False ▷ Make sure this is an inverted sentence
7:     actual-verb ← t
8:     be-verb ← t.head
9:     return True
10: return False

Algorithm 11 Transform() in Inverted Clausal Subject Analyzer
1: procedure INVERTED_CLAUSAL_SUBJECT TRANSFORM Procedure(sents) ▷ Input sentence
2: get access to actual-verb, be-verb and attr in identify procedure 10
3: change position of actual-verb and attr in be-verb’s children
4: return [traverse_a_string(be-verb)]
Algorithm 12 Transform() in CCOMP Analyzer

```
procedure Clause Component transform Procedure(sents) ▷ Input sentence
1: cut CCOMP link in dependency tree
2: subject ← find ccomp verb’s subject
3: if subject ≠ NULL and subject ≠ DET(e.g. ’that’) then
4: if original verb do not have object then
5: make subject as original verb’s object
6: else if subject = DET(e.g. ‘that’) then
7: find root verb’s object, substitute ‘that’
8: str1 ← traverse_a_string(ccomp verb)
9: str2 ← traverse_a_string(original verb)
10: return [str1, str2]
```

Algorithm 13 Transform() in XCOMP Analyzer

```
procedure Open Clausal Component transform Procedure(sents) ▷ Input sentence
1: subject-token ← find subject(xcomp-verb-token) ▷ find subject of the actual verb
2: results ← empty list
3: if AUX ∉ sents then ▷ for some cases two verbs needs to be output
4: cut xcomp link
5: results.add(traverse_a_string(xcomp-verb-token))
6: remove AUX token in the dependency tree
7: replace xcomp-verb-token in subject’s children with actual-verb-token
8: results.add(traverse_a_string(actual-verb-token))
9: return results
```

Algorithm 14 Transform() in ACL Analyzer

```
procedure Adjective Clause transform Procedure(sents) ▷ Input Sentence
1: cut acl edge in dependency tree
2: str1 ← traverse_a_string(root-token)
3: mid-fix ← empty string
4: if acl-token.tag = VBN and by-token in acl-token’s right children then
5: mid-fix ← ’be’
6: update acl-verb-token’s left children with [acl-noun, mid-fix, [t ∈ acl-verb-token.lefts where t.dep ∉ [AUX]]
7: correct acl-verb-tense
8: str2 ← traverse_a_string(acl-verb-token)
9: return [str1, str2]
```

Algorithm 15 Get-Temporal Function at Line 5 in Algorithm 3

```
procedure Adverbial Clause Temporal Info Extraction Procedure(prep – or – mark, cur – temp): The PREP or MARK token in input sentence
1: flag ← False ▷ whether we change the temp
2: if prep-or-mark.type = PREP then
3: sign ← -1
4: else
5: sign ← 1
6: text ← prep-or-mark.text.lower()
7: if text = as then
8: flag ← True
9: else if text ∈ [until, till] then
10: flag ← True
11: cur-temp ← cur-temp + sign
12: else if text = after then
13: flag ← True
14: cur-temp ← cur-temp - sign
15: else if text = before then
16: flag ← True
17: cur-temp ← cur-temp + sign
18: return [flag, cur-temp]
```
Table 8: Example model outputs. We compare against YATS [Ferrés et al., 2016] and an end-to-end sentence simplification system [Vu et al., 2018].
Appendix B: Action Representation Fields

Action Representation Fields (ARFs) in the demo sentence *James gently throws a red ball to Alice in the restaurant from back*, extracted with SRL:

- **owner**: James
- **target**: a red ball
- **prop**: to Alice
- **action**: throw
- **origin_action**: throws
- **manner**: gently
- **modifier_location**: in the restaurant
- **modifier_direction**: from back

In this case, our output for the prop and target is not correct; they should be swapped. This is one example where this module can introduce errors.

Additional ARFs, extracted heuristically:

- **startTime**: Calculated by current scene time
- **duration**: We have a pre-defined list of words that when appearing in the sentence, they will indicate a short duration (e.g. “run” or “fast”) and therefore the duration will be set to 1 second; in contrast, for words like “slowly” we assign a duration of 4 seconds; otherwise, the duration is 2 seconds.
- **speed**: Similarly to duration, we have pre-defined lists of words that would affect the speed of the pre-baked animation: “angrily” would result in faster movement, but “carefully” in slower movement. We have 3 scales: 0.5, 1, 2 which corresponds to slow, normal and fast.
- **translation**: We have a list of actions which would entail a movement in from one place to another, e.g. “go”. If the value of the action exists in this list, it is set to True, otherwise False.
- **rotation**: If the action exists in our list of verbs that entail rotation, this field is True, otherwise False. Rotation refers to movement in place e.g. “turn” or “sit”.
- **emotion**: We find the closet neighbor of each word in the sentence in list of words that indicate emotion, using word vector similarity. If the similarity exceeds an empirically tested threshold, then we take the corresponding emotion word as the emotion field of this action.
- **partial_start_time**: an important field, since it controls the sequence order of each action. It determines which actions happen in parallel and which happen sequentially. This is still an open question. We solve this problem when doing sentence simplification. Together with the input sentence, current time is also fed into each analyzer. There are several rules in some of the analyzers to obtain temporal information. For example, in Line 5 of the adverbial clause analyzer (c.f.3), we assign different temporal sequences for simplified actions. The algorithm is shown in Algorithm 15. The sign together with specific prepositions determines the change direction of current temporal id. In the Coordination Analyzer, the current temporal id changes when it encounters two verbs sharing same subject. Then the later action will get a bigger temporal id.